
Dive into
Deep Learning
ADVANCED
Alexander J Smola · Rasool Fakoor

Table of contents

Advanced **1**

9 Optimization Algorithms **3**
Resources and Further Reading 4
9.1 Landscapes 5
9.1.1 The Goal of Optimization 5
9.1.2 Critical Points and Plateaus 6
9.1.3 Curvature and Noise 10
9.1.4 The Role of Convexity 14
9.1.5 Three Decisions 15
9.1.6 Summary 15
9.1.7 Exercises 15
9.2 Gradient Descent 17
9.2.1 One-Dimensional Gradient Descent 17
9.2.2 Multivariate Gradient Descent 21
9.2.3 Newton's Method 23
9.2.4 Preconditioning 26
9.2.5 Summary 27
9.2.6 Exercises 28
9.3 Stochastic and Minibatch Gradient Descent 29
9.3.1 Stochastic Gradient Descent 30
9.3.2 Minibatches 32
9.3.3 Minibatch SGD on the Airfoil Data 37
9.3.4 Summary 42
9.3.5 Exercises 43
9.4 Momentum 44
9.4.1 An Ill-Conditioned Valley 45
9.4.2 The Momentum Method 46
9.4.3 Implementation 50
9.4.4 Nesterov Momentum 54
9.4.5 Summary 56
9.4.6 Exercises 57
9.5 Adam and AdamW 58
9.5.1 Per-Coordinate Learning Rates 58
9.5.2 TinyLM and FashionCNN 64
9.5.3 Strengths and Limits of Adam 70
9.5.4 Decoupled Weight Decay 75

9.5.5 Optimizer-State Memory . . . 83
9.5.6 Summary . . . 84
9.5.7 Exercises . . . 84
9.6 Schedules . . . 87
9.6.1 Decay on the Valley . . . 87
9.6.2 Decay on a Network . . . 90
9.6.3 Warmup . . . 96
9.6.4 Warmup–Stable–Decay . . . 98
9.6.5 Summary . . . 102
9.6.6 Exercises . . . 102
9.7 Muon . . . 104
9.7.1 Norms and Steepest Descent . . . 105
9.7.2 Orthogonalization by Newton–Schulz . . . 107
9.7.3 Muon from Scratch . . . 111
9.7.4 Relatives and Variants . . . 118
9.7.5 Summary . . . 121
9.7.6 Exercises . . . 121
9.8 Batch Size . . . 123
9.8.1 The Gradient-Noise Scale . . . 124
9.8.2 Steps to a Target . . . 127
9.8.3 Learning-Rate Rules . . . 132
9.8.4 Growing the Batch . . . 136
9.8.5 Summary . . . 136
9.8.6 Exercises . . . 136
9.9 Scaling Up . . . 138
9.9.1 Learning Rate and Width . . . 138
9.9.2 Maximal Update Parametrization . . . 141
9.9.3 The Spectral View . . . 146
9.9.4 Summary . . . 147
9.9.5 Exercises . . . 147
9.10 Practice . . . 149
9.10.1 Choosing a Configuration . . . 149
9.10.2 Gradient Clipping . . . 151
9.10.3 Weight Averaging . . . 154
9.10.4 Comparing Optimizers . . . 157
9.10.5 Summary . . . 158
9.10.6 Exercises . . . 158
9.11 Further Reading . . . 160
9.11.1 The Loss Surface . . . 160
9.11.2 Classical Methods . . . 161
9.11.3 Adaptive and Matrix Preconditioners . . . 162
9.11.4 Schedules and Averages . . . 164
9.11.5 Large Runs and Benchmarks . . . 165

10 Attention **167**
Resources and Further Reading . . . 168
10.1 Queries, Keys, and Values . . . 169
10.1.1 Attention as Differentiable Lookup . . . 169
10.1.2 Attention Pooling with Fixed Kernels . . . 170
10.1.3 Attention Weights as a Diagnostic . . . 173

10.1.4 Summary . . . 174
10.1.5 Exercises . . . 174
10.2 Attention Scoring and Masking . . . 176
10.2.1 Dot-Product Attention . . . 176
10.2.2 Masking . . . 179
10.2.3 Batched Attention . . . 184
10.2.4 Attention as Learned Alignment . . . 186
10.2.5 Summary . . . 188
10.2.6 Exercises . . . 189
10.3 Multi-Head and Cross-Attention . . . 191
10.3.1 A One-Mixture Bottleneck with Position-Only Keys . . . 191
10.3.2 Multi-Head Attention . . . 193
10.3.3 Self-Attention and Cross-Attention . . . 199
10.3.4 Summary . . . 201
10.3.5 Exercises . . . 202
10.4 Positional Information . . . 205
10.4.1 Permutation Equivariance . . . 205
10.4.2 Absolute Position Embeddings . . . 206
10.4.3 Rotary Position Embeddings . . . 209
10.4.4 Extrapolation Beyond the Training Length . . . 211
10.4.5 Summary . . . 216
10.4.6 Exercises . . . 216
10.5 The Cost of Attention . . . 219
10.5.1 Quadratic Time and Memory . . . 220
10.5.2 Exact Attention Without the Matrix . . . 223
10.5.3 Windowed and Sparse Attention . . . 227
10.5.4 Linear Attention in Recurrent Form . . . 230
10.5.5 Measured Running Time . . . 233
10.5.6 Summary . . . 234
10.5.7 Exercises . . . 235
10.6 Attention Circuits . . . 238
10.6.1 The Residual Stream . . . 238
10.6.2 Copying and Induction . . . 241
10.6.3 Evidence for the Circuit . . . 245
10.6.4 In-Context Learning . . . 250
10.6.5 Attention Weights as Explanations . . . 252
10.6.6 Summary . . . 254
10.6.7 Exercises . . . 254
10.7 Further Reading . . . 257
10.7.1 The Lookup and Its Normalization . . . 258
10.7.2 Heads . . . 259
10.7.3 Positions . . . 259
10.7.4 Cost . . . 260
10.7.5 Circuits in Large Models . . . 262

11 Transformers **265**
Resources and Further Reading . . . 266
11.1 The Transformer Block . . . 267
11.1.1 Components of a Transformer Block . . . 267
11.1.2 Normalization Placement . . . 268

11.1.3 The Feed-Forward Network . . . 273
11.1.4 A Configurable Block . . . 274
11.1.5 Summary . . . 277
11.1.6 Exercises . . . 278
11.2 A GPT from Scratch . . . 280
11.2.1 From Blocks to a Language Model . . . 281
11.2.2 Training the Model . . . 286
11.2.3 Sampling from the Model . . . 290
11.2.4 Loading GPT-2 . . . 291
11.2.5 Summary . . . 294
11.2.6 Exercises . . . 294
11.3 Generation and the KV Cache . . . 298
11.3.1 From Recomputation to Caching . . . 298
11.3.2 Memory and Bandwidth . . . 303
11.3.3 Sharing Keys and Values across Heads . . . 307
11.3.4 Compressing the Cache Further . . . 308
11.3.5 Summary . . . 313
11.3.6 Exercises . . . 313
11.4 Encoders, Decoders, and Cross-Attention . . . 316
11.4.1 Encoder, Decoder, and Encoder–Decoder Architectures . . . 317
11.4.2 Bidirectional Context for Masked-Token Prediction . . . 318
11.4.3 Cross-Attention in an Encoder–Decoder . . . 321
11.4.4 Cross-Attention as Interface . . . 327
11.4.5 Summary . . . 329
11.4.6 Exercises . . . 330
11.5 Vision Transformer . . . 332
11.5.1 Patches as Tokens . . . 333
11.5.2 The Vision Transformer . . . 334
11.5.3 Training on Fashion-MNIST . . . 337
11.5.4 Comparison with a Convolutional Network . . . 340
11.5.5 Summary . . . 343
11.5.6 Exercises . . . 343
11.6 Mixture of Experts . . . 346
11.6.1 Conditional Computation . . . 347
11.6.2 A Mixture-of-Experts Layer . . . 348
11.6.3 Routing Collapse and Load Balancing . . . 352
11.6.4 Mixture-of-Experts Language Models . . . 356
11.6.5 Summary . . . 360
11.6.6 Exercises . . . 360
11.7 Scaling Laws . . . 363
11.7.1 Counting Parameters and FLOPs . . . 364
11.7.2 A Miniature Scaling Study . . . 366
11.7.3 Compute-Optimal Allocation . . . 368
11.7.4 Configurations of Deployed Models . . . 372
11.7.5 Summary . . . 374
11.7.6 Exercises . . . 374
11.8 Further Reading . . . 376
11.8.1 The Block . . . 377
11.8.2 The Training Recipe . . . 378
11.8.3 Generation and the Cache . . . 378

11.8.4 Wirings and Images . . . 379
11.8.5 Experts . . . 381
11.8.6 Scaling . . . 382

12 State Space Models **383**
Resources and Further Reading . . . 384
12.1 Gated Recurrent Networks . . . 385
12.1.1 Gates . . . 385
12.1.2 Long Short-Term Memory . . . 387
12.1.3 Gated Recurrent Units . . . 389
12.1.4 Training Gated Networks . . . 390
12.1.5 Summary . . . 396
12.1.6 Exercises . . . 397
12.2 Linear Recurrences and State Space Models . . . 399
12.2.1 Linear Recurrences . . . 399
12.2.2 Parallel Scans . . . 401
12.2.3 State Space Models . . . 406
12.2.4 S4D . . . 409
12.2.5 Summary . . . 412
12.2.6 Exercises . . . 412
12.3 Selective State Space Models . . . 414
12.3.1 The Selectivity Problem . . . 415
12.3.2 Selective Dynamics . . . 416
12.3.3 The Mamba Layer . . . 418
12.3.4 Summary . . . 424
12.3.5 Exercises . . . 424
12.4 Matrix-State Recurrences . . . 425
12.4.1 An Associative Memory . . . 426
12.4.2 Decay . . . 429
12.4.3 The Dual Form . . . 430
12.4.4 A Gated Linear Attention Layer . . . 434
12.4.5 Summary . . . 437
12.4.6 Exercises . . . 437
12.5 The Delta Rule . . . 439
12.5.1 Additive Writes . . . 439
12.5.2 Writes as Gradient Steps . . . 440
12.5.3 Gated DeltaNet . . . 442
12.5.4 Memory as Regression . . . 446
12.5.5 Summary . . . 447
12.5.6 Exercises . . . 447
12.6 Hybrid Architectures . . . 449
12.6.1 Limits of a Fixed-Size State . . . 449
12.6.2 Recall with One Attention Layer . . . 451
12.6.3 Inference Memory . . . 454
12.6.4 Summary . . . 456
12.6.5 Exercises . . . 456
12.7 Further Reading . . . 457
12.7.1 State Space Models . . . 457
12.7.2 Writing to a Matrix Memory . . . 459
12.7.3 Recurrence and Attention Together . . . 459

12.7.4 Beyond Language 460

13 Computational Performance **463**
Resources and Further Reading 464
13.1 The Performance Model 465
13.1.1 Arithmetic Intensity and the Roofline 465
13.1.2 Timing GPU Code 468
13.1.3 Three Regimes 470
13.1.4 Profiling a Training Step 474
13.1.5 Summary 476
13.1.6 Exercises 476
13.2 Hardware 478
13.2.1 The Memory Hierarchy 479
13.2.2 Compute and Memory Bandwidth 482
13.2.3 The GPU 484
13.2.4 Data Paths outside the GPU 486
13.2.5 Workloads on the Roofline 488
13.2.6 Summary 490
13.2.7 Exercises 490
13.3 Compute Graphs and Compilation 494
13.3.1 Graph Capture 494
13.3.2 Operation Fusion 497
13.3.3 Compiling a Training Step 500
13.3.4 Kernel Replay 501
13.3.5 Limits of Compilation 502
13.3.6 Summary 503
13.3.7 Exercises 503
13.4 Memory and Precision 505
13.4.1 Memory of a Training Step 506
13.4.2 Mixed Precision 512
13.4.3 Activation Checkpointing 513
13.4.4 Gradient Accumulation 515
13.4.5 Choosing a Memory Technique 516
13.4.6 Summary 516
13.4.7 Exercises 517
13.5 Multi-GPU from First Principles 519
13.5.1 Three Ways to Split 520
13.5.2 Data Parallelism by Hand 521
13.5.3 Ring Allreduce 525
13.5.4 Communication Cost and Scaling 527
13.5.5 Summary 531
13.5.6 Exercises 531
13.6 Multi-GPU in Practice 533
13.6.1 Production Data Parallelism 534
13.6.2 Scaling on Four GPUs 539
13.6.3 Sharding the Training State 543
13.6.4 Summary 545
13.6.5 Exercises 545
13.7 Case Study: Optimizing a Transformer 548
13.7.1 Experimental Setup 549

13.7.2 A Single GPU . . . 551
13.7.3 Data Parallelism . . . 556
13.7.4 Results . . . 558
13.7.5 Summary . . . 560
13.7.6 Exercises . . . 560
13.8 Further Reading . . . 562
13.8.1 Measuring and Modeling . . . 562
13.8.2 Hardware and Number Formats . . . 564
13.8.3 Compilers and Kernels . . . 565
13.8.4 Memory . . . 566
13.8.5 Communication . . . 567
13.8.6 Beyond One Machine . . . 568

14 Reinforcement Learning **569**
Resources and Further Reading . . . 570
14.1 Markov Decision Processes . . . 571
14.1.1 The Model . . . 574
14.1.2 Return, Discount and Horizon . . . 575
14.1.3 The Choice of State . . . 579
14.1.4 Reward Design . . . 581
14.1.5 Two Special Cases . . . 581
14.1.6 Summary . . . 584
14.1.7 Exercises . . . 584
14.2 Dynamic Programming . . . 587
14.2.1 Values and the Bellman Equations . . . 587
14.2.2 Value Iteration . . . 590
14.2.3 Policy Evaluation and Policy Iteration . . . 598
14.2.4 The Optimal Policy on Slippery Ice . . . 600
14.2.5 Reward Shaping and the Optimal Policy . . . 604
14.2.6 Summary . . . 606
14.2.7 Exercises . . . 606
14.3 Temporal-Difference Learning . . . 610
14.3.1 The Sampled Backup . . . 611
14.3.2 Q-Learning on the Slippery Lake . . . 615
14.3.3 Off-Policy Learning . . . 621
14.3.4 Planning in a Learned Model . . . 622
14.3.5 Summary . . . 624
14.3.6 Exercises . . . 624
14.4 Exploration . . . 627
14.4.1 The Multi-Armed Bandit . . . 628
14.4.2 Optimism and Posterior Sampling . . . 631
14.4.3 Exploration in MDPs . . . 635
14.4.4 Exploration on the Verifier Task . . . 639
14.4.5 Summary . . . 641
14.4.6 Exercises . . . 641
14.5 Learning from Demonstrations . . . 643
14.5.1 Behavior Cloning . . . 644
14.5.2 Distribution Shift . . . 648
14.5.3 Compounding Error . . . 651
14.5.4 Interactive Data Collection . . . 653

14.5.5 Summary . . . 656
14.5.6 Exercises . . . 657
14.6 Policy Gradient . . . 660
14.6.1 Parameterizing the Policy . . . 660
14.6.2 The Policy Gradient . . . 661
14.6.3 REINFORCE on the Running Examples . . . 664
14.6.4 The Policy Gradient Theorem . . . 670
14.6.5 Costs and Limitations . . . 671
14.6.6 Summary . . . 674
14.6.7 Exercises . . . 674
14.7 Baselines and Variance Reduction . . . 677
14.7.1 The Zero-Mean Identity . . . 678
14.7.2 Reward-to-Go, Baselines, and Control Variates . . . 679
14.7.3 Centering, Scaling, and Normalization . . . 683
14.7.4 Five Estimators Compared . . . 688
14.7.5 Summary . . . 694
14.7.6 Exercises . . . 695
14.8 Function Approximation in Reinforcement Learning . . . 699
14.8.1 Networks in Place of Tables . . . 700
14.8.2 REINFORCE on CartPole . . . 703
14.8.3 Continuous Actions . . . 706
14.8.4 Generalization across States . . . 711
14.8.5 A Map of the Methods . . . 713
14.8.6 Summary . . . 715
14.8.7 Exercises . . . 715

15 Deep Reinforcement Learning **721**
Resources and Further Reading . . . 722
15.1 Actor-Critic Methods and Multi-Step Returns . . . 722
15.1.1 The Temporal-Difference Advantage . . . 723
15.1.2 The Actor-Critic Update . . . 725
15.1.3 Multi-Step Returns and GAE . . . 732
15.1.4 Bias and Variance of the λ-Return . . . 734
15.1.5 Summary . . . 739
15.1.6 Exercises . . . 739
15.2 Trust Regions and Proximal Policy Optimization . . . 743
15.2.1 Parameter Space versus Policy Space . . . 743
15.2.2 Importance Sampling . . . 744
15.2.3 Trust Regions . . . 746
15.2.4 PPO in Practice . . . 749
15.2.5 Summary . . . 760
15.2.6 Exercises . . . 761
15.3 KL-Regularized Policy Optimization . . . 765
15.3.1 Learning a Reward from Comparisons . . . 765
15.3.2 Optimizing a Fitted Reward . . . 769
15.3.3 The KL-Regularized Objective . . . 769
15.3.4 The Soft Backup . . . 774
15.3.5 Summary . . . 778
15.3.6 Exercises . . . 778
15.4 Deep Q-Networks . . . 781

15.4.1 Sources of Instability . . . 782
15.4.2 Replay and Target Networks . . . 784
15.4.3 Overestimation . . . 794
15.4.4 Summary . . . 797
15.4.5 Exercises . . . 798
15.5 Soft Actor-Critic . . . 802
15.5.1 The Maximum-Entropy Objective . . . 803
15.5.2 The Squashed Gaussian Policy . . . 806
15.5.3 The Algorithm . . . 809
15.5.4 Sample Efficiency and Calibration . . . 815
15.5.5 Summary . . . 822
15.5.6 Exercises . . . 822
15.6 Learned Models and Planning . . . 826
15.6.1 A Dynamics Model from Data . . . 827
15.6.2 Planning in the Model . . . 832
15.6.3 Search as Policy Improvement . . . 836
15.6.4 Summary . . . 840
15.6.5 Exercises . . . 841
15.7 Offline Reinforcement Learning . . . 842
15.7.1 Learning from a Fixed Record . . . 843
15.7.2 Pessimism . . . 850
15.7.3 Beyond the Table . . . 853
15.7.4 Summary . . . 854
15.7.5 Exercises . . . 854
15.8 Reinforcement Learning for Sequence Generation . . . 857
15.8.1 A Response as a Trajectory . . . 858
15.8.2 Verifiers, Reward Models and Their Loopholes . . . 859
15.8.3 The Post-Training Update . . . 861
15.8.4 Summary . . . 864
15.8.5 Exercises . . . 864
15.9 Further Reading . . . 866
15.9.1 Policy Optimization in Practice . . . 866
15.9.2 Value Learning after DQN . . . 867
15.9.3 Continuous Control since 2018 . . . 868
15.9.4 Models, Imagination and Search . . . 868
15.9.5 Offline Methods at Scale . . . 869
15.9.6 Post-Training Variants . . . 869
15.9.7 Beyond the Chapter . . . 870
15.9.8 A Project . . . 870

16 Generative Adversarial Networks **873**
Resources and Further Reading . . . 873
16.1 Generative Adversarial Networks . . . 874
16.1.1 From Likelihood to Comparison . . . 874
16.1.2 The Log-Loss Game . . . 877
16.1.3 The Generator's Gradient . . . 879
16.1.4 The Game on Fashion-MNIST . . . 881
16.1.5 The Game on a Gaussian . . . 884
16.1.6 Summary . . . 890
16.1.7 Exercises . . . 890

16.2 Adversarial Objectives and Divergences . . . 893
16.2.1 A Common Template and Two Choices . . . 893
16.2.2 Classification Losses and Their Divergences . . . 894
16.2.3 Integral Probability Metrics . . . 899
16.2.4 Gradients Far from and Near the Data . . . 901
16.2.5 Summary . . . 905
16.2.6 Exercises . . . 906
16.3 Relativistic Objectives . . . 908
16.3.1 Scoring Pairs . . . 908
16.3.2 The Value of the Pairing Game . . . 909
16.3.3 Ranking and Mode Coverage . . . 914
16.3.4 Summary . . . 919
16.3.5 Exercises . . . 919
16.4 Gradient Penalties and Convergence . . . 921
16.4.1 The Dirac-GAN . . . 922
16.4.2 Zero-Centered Penalties . . . 923
16.4.3 The Penalized Value Near Equilibrium . . . 926
16.4.4 The R3GAN Recipe . . . 929
16.4.5 The Games of This Chapter . . . 933
16.4.6 Summary . . . 934
16.4.7 Exercises . . . 934
16.5 Adversarial Image Generation . . . 936
16.5.1 The 2015 Recipe . . . 937
16.5.2 A Backbone without Normalization . . . 941
16.5.3 A Controlled Loss Comparison . . . 945
16.5.4 Measuring Sample Quality . . . 949
16.5.5 Limited Data . . . 952
16.5.6 Summary . . . 953
16.5.7 Exercises . . . 953
16.6 Conditional Generation . . . 956
16.6.1 The Conditional Game . . . 956
16.6.2 Conditioning Mechanisms . . . 958
16.6.3 Class-Conditional CIFAR-10 . . . 959
16.6.4 Measuring Condition Alignment . . . 963
16.6.5 Summary . . . 966
16.6.6 Exercises . . . 967
16.7 Adversarial Losses Beyond GANs . . . 969
16.7.1 Conditional Means and Ambiguity . . . 969
16.7.2 Adversarial Terms in Current Systems . . . 973
16.7.3 The Objective without the Game . . . 974
16.7.4 Summary . . . 976
16.7.5 Exercises . . . 976
16.8 Further Reading . . . 979
16.8.1 Objectives and Critic Constraints . . . 979
16.8.2 Architectures and Training Devices . . . 980
16.8.3 Evaluation . . . 981
16.8.4 Conditions, Translation and Text . . . 982
16.8.5 Fixed Discrepancies and Learned Ones . . . 983
16.8.6 Adversarial Terms in Recent Systems . . . 983

17 Diffusion Models **985**
Resources and Further Reading . . . 986
17.1 Denoising and the Score . . . 986
17.1.1 The Running Example . . . 987
17.1.2 The Optimal Denoiser . . . 990
17.1.3 Limits of a Single Noise Level . . . 995
17.1.4 Denoising Images . . . 999
17.1.5 Summary . . . 1001
17.1.6 Exercises . . . 1001
17.2 Gaussian Paths: Diffusion and Flow Matching . . . 1005
17.2.1 One Path from Data to Noise . . . 1005
17.2.2 One Regression . . . 1009
17.2.3 Sampling by Integration . . . 1015
17.2.4 Experiments on the Running Example . . . 1018
17.2.5 Summary . . . 1021
17.2.6 Exercises . . . 1021
17.3 Diffusion for Images . . . 1024
17.3.1 The Denoising U-Net . . . 1025
17.3.2 The Training Recipe . . . 1028
17.3.3 Measuring Sample Quality . . . 1030
17.3.4 Guidance . . . 1033
17.3.5 Summary . . . 1036
17.3.6 Exercises . . . 1036
17.4 Samplers, Inversion and Editing . . . 1040
17.4.1 One Field, Two Samplers . . . 1040
17.4.2 Fewer Steps . . . 1043
17.4.3 Inversion and Editing . . . 1049
17.4.4 Summary . . . 1053
17.4.5 Exercises . . . 1053
17.5 Likelihood, Weighting and the Design Space . . . 1056
17.5.1 The Likelihood Bound . . . 1057
17.5.2 Every Objective Is a Weighting . . . 1062
17.5.3 The Design Space . . . 1064
17.5.4 Summary . . . 1070
17.5.5 Exercises . . . 1070
17.6 Latent Diffusion and Transformers . . . 1073
17.6.1 Why a Latent Space . . . 1074
17.6.2 Diffusion in the Latent Space . . . 1077
17.6.3 The Transformer Backbone . . . 1080
17.6.4 Conditioning on Text and Beyond . . . 1086
17.6.5 Summary . . . 1086
17.6.6 Exercises . . . 1086
17.7 Few-Step Generation . . . 1088
17.7.1 Why One Step Fails . . . 1089
17.7.2 Flow Maps . . . 1090
17.7.3 Experiments . . . 1095
17.7.4 The Other Route: Matching Distributions . . . 1100
17.7.5 Summary . . . 1100
17.7.6 Exercises . . . 1101
17.8 Discrete Diffusion . . . 1103

17.8.1 Corrupting Tokens . . . 1103
17.8.2 The Masked Bound . . . 1105
17.8.3 Sampling by Unmasking . . . 1107
17.8.4 Experiments . . . 1110
17.8.5 Masked Language Models as One Step . . . 1116
17.8.6 Summary . . . 1117
17.8.7 Exercises . . . 1117
17.9 Further Reading . . . 1121
17.9.1 Energy-Based Models and Score Matching . . . 1121
17.9.2 Schedules, Weighting and Networks . . . 1122
17.9.3 Samplers . . . 1123
17.9.4 Guidance, Editing and Inverse Problems . . . 1123
17.9.5 Latent Spaces and Backbones . . . 1123
17.9.6 Few-Step Generation . . . 1124
17.9.7 Likelihood . . . 1124
17.9.8 Evaluation, Memorization and Generalization . . . 1124
17.9.9 Video . . . 1124
17.9.10 Discrete Diffusion and Diffusion Language Models . . . 1124
17.9.11 Post-Training by Reward and Preference . . . 1125

References **1127**

Advanced

The Basics ended with a small recurrent network trained on *The Time Machine*. Asked to continue the phrase "The Time Traveller", it writes real words and punctuation, but its grammar holds over a few words at most. The language models that answer questions and write programs today are built from the same parts: layers, a loss, its gradient and an optimizer. What does it take to get from one to the other? This part treats four problems that the small network never met.

The first is training. A learning rate tuned for one model may make a model ten times wider diverge. The second is memory. A recurrent network squeezes everything it has read into a state of fixed size, written before it knows what will be asked. In 2014 attention let a translation model consult every source word. In 2017 Ashish Vaswani and colleagues at Google built a translation model from attention alone, without recurrence. That model, the transformer, underlies most large language models built since.

The third is hardware. A GPU computes much faster than it moves data, so a model's cost depends on how its operations use memory, not only on how many it performs. The fourth is the objective. An agent that plays a game learns from rewards rather than labels. A model that generates images has no label to match, so its loss must judge whether a new sample could have come from the data.

Chapter 9 asks how gradient methods behave on curved, noisy losses, and how to choose a learning rate, schedule and batch size. Chapter 10 asks how a position can retrieve what it needs from any other. Chapter 11 asks how attention and feed-forward layers stack into a deep model that trains reliably. Chapter 12 asks how a state of fixed size can be trained on all positions at once, and what it can store. Chapter 13 asks what limits the speed of a training step on one GPU and on several. Chapter 14 asks how an agent learns when its actions change the data it sees, and Chapter 15 asks how to keep that learning stable with neural networks. Chapter 16 asks how to train a generator that offers samples but no density, and Chapter 17 asks how to generate data by learning to remove noise.

The Advanced chapters are implemented in PyTorch and JAX. The Language Models and Image Models parts apply them to text and images.

Chapter 9

Optimization Algorithms

Every network in this book was trained with optimizer settings taken from a recipe, such as SGD with momentum 0.9 or AdamW with warmup and cosine decay (Section 7.6). When a loss shoots up to `nan`, we halve the learning rate and try again. Why does a larger step ruin a run, and why does a smaller one cost so much time? What does each part of the recipe do, and what must change when the model grows?

Gradient descent is older than computers. Augustin-Louis Cauchy proposed it in 1847 to solve the equations that determine the orbit of a celestial body. In 1951 Herbert Robbins and Sutton Monro asked how to find the root of a function that can only be measured with noise, as a minibatch gradient is, and answered with steps whose size shrinks over time. Two problems limit such steps. The curvature of the loss differs between directions, so a step that is safe along the steepest one barely moves along the flattest. And the gradient is only an estimate, so every step is slightly wrong. Momentum, per-coordinate learning rates, schedules and batch sizes each answer one of these problems. Newer methods also ask how to measure the size of a step. We take the questions one at a time, on problems we can solve exactly.

Section 9.1 asks what makes a loss hard to minimize. Section 9.2 asks how large a gradient step may be, and how rescaling the gradient removes the limit. Section 9.3 asks what changes when the gradient is estimated from a few examples. Section 9.4 asks how averaging gradients over steps speeds up descent along a narrow valley. Section 9.5 asks what a separate learning rate for each coordinate achieves, and why weight decay must then bypass it. Section 9.6 asks how the learning rate should change over a run. Section 9.7 asks what step a weight matrix should take when the size of a step is measured as a matrix. Section 9.8 asks how large a batch can grow before more examples per step stop saving steps. Section 9.9 asks how learning rates carry over from a narrow model to a wide one. Section 9.10 asks how to configure a long run, guard it against failure, and compare optimizers fairly.

The mathematical appendix (Chapter 35) proves the rates that this chapter measures, and Chapter 13 spreads the same updates across many devices. Methods the chapter does not teach are listed in Section 9.11.

Resources and Further Reading

The resources below accompany the sections of this chapter; each names the section it goes with. All are freely accessible online except where noted. Related topics that the chapter does not teach are collected in Section 9.11, and the optimization chapter of the mathematical appendix (Chapter 35) keeps its own list for the theory, with convex-optimization texts and courses.

Books

- Convex Optimization — Boyd & Vandenberghe — free PDF; the standard reference for the vocabulary this chapter uses informally (conditioning, convergence rates, projections), and the place where the analyses that Section 9.2 and Section 9.3 state without proof are carried out in full.
- Numerical Optimization — Nocedal & Wright — paywalled, widely held in university libraries; a comprehensive treatment of Newton's method and preconditioning (Section 9.2) and of the methods built on them.

Courses and video lectures

- Stanford CS336: Language Modeling from Scratch — Assignment 1 — free; the graded version of several exercises of this chapter: implement AdamW as Section 9.5.4 does, account for optimizer-state memory byte by byte, and run the learning-rate and batch-size sweeps that Section 9.10 turns into a method; the accompanying lectures are on YouTube.

Foundational and current papers

- Old Optimizer, New Norm: An Anthology — Bernstein & Newhouse (2024) — free; the unification that organizes Section 9.7: SGD, sign descent and Adam, and Shampoo are each steepest descent under a different norm.
- An Empirical Model of Large-Batch Training — McCandlish et al. (2018) — free; defines the gradient-noise scale and the critical batch size, the two quantities measured in Section 9.8, and predicts when doubling the batch stops halving the steps.
- Understanding Warmup-Stable-Decay Learning Rates: A River Valley Loss Landscape Perspective — Wen et al. (2024) — free; the river-valley landscape of Section 9.6, in which the stable phase travels along the river and the decay phase descends its bank; this explains why the loss drops when a warmup–stable–decay schedule begins to decay.
- Fantastic Pretraining Optimizers and Where to Find Them — Wen et al. (2025) — free; re-benchmarks ten optimizers under matched tuning and finds that many reported speedups over AdamW shrink, the evidence behind the matched comparisons of Section 9.7 and Section 9.10.

Tutorials, notes, and interactive

- Why Momentum Really Works — Gabriel Goh, Distill (2017) — free, interactive; damping and acceleration with sliders for η and β, showing the critical value of β and the ringing trajectories of Section 9.4.
- An Overview of Gradient Descent Optimization Algorithms — Sebastian Ruder (2016) — free; a survey of the classical progression from Section 9.3 through Section 9.5, and a record of common practice in 2016.
- Deep Learning Tuning Playbook — Godbole et al., Google Research — free; the division into scientific, nuisance, and fixed hyperparameters and the budget-tiered sweeps

that Section 9.10 teaches, from a team that ran them at production scale.
- Muon: An Optimizer for Hidden Layers in Neural Networks — Keller Jordan (2024) — free; the original post, with the design decisions, Newton–Schulz coefficients, and ablations behind the optimizer that Section 9.7 builds from scratch.
- Deriving Muon — Jeremy Bernstein — free; a compact derivation of Muon as steepest descent under a norm, the note-form companion to the derivation in Section 9.7.
- The Practitioner's Guide to the Maximal Update Parameterization — EleutherAI — free; muP implemented step by step with the coordinate checks of Section 9.9, including the failures a first implementation runs into.

9.1 Landscapes

Training a network is a walk downhill. A model with a million parameters defines a loss over a million-dimensional space, and every optimizer in this chapter does the same thing on that surface: it measures the slope where it stands and takes a step. Whether the step makes progress depends on the shape of the surface nearby. Where the surface is flat, the slope carries almost no information and the step goes nowhere. Where it curves steeply in one direction and gently in another, a step small enough to be safe in the steep direction barely moves along the gentle one. And because the slope is computed from a minibatch rather than from the whole dataset, every step is also slightly wrong.

This section examines these shapes before we build any optimizer. We first ask what minimizing the training loss achieves and what it cannot achieve. We then look at the points where the slope vanishes: local minima, saddle points, and plateaus. The larger part of the section is about two effects that slow routine training far more often than critical points do: uneven curvature and noisy gradients. We measure the first on a two-dimensional valley that the rest of the chapter reuses, and we watch a small network adjust its own curvature to the step size. Convex analysis, the source of most of our vocabulary, closes the section.

9.1.1 The Goal of Optimization

In optimization, the function to be minimized is called the *objective function*. By convention we minimize; maximizing a quantity is the same as minimizing its negative. Optimization texts also collect all parameters into a single vector $\mathbf{x}$, and we follow them in this chapter: $\mathbf{x}$ holds everything that earlier chapters wrote as $(\mathbf{w}, b)$.

Optimization and learning share a loss function but not a goal. The optimizer minimizes the *empirical risk*, the average loss over the training set. Learning wants a low *risk*, the expected loss over the whole population of examples the model will ever see (Section 2.6.1). The optimizer only ever sees the former, and as Section 2.6 showed, driving the training error toward zero can even raise the error on new data.

A one-dimensional example shows the difference. Let f be a smooth risk and let g be an empirical risk that fluctuates around it, as a finite-sample average fluctuates around its expectation. The minimum of g need not sit at the minimum of f, and in Figure 9.1 it does not.

No optimizer can close this gap, because it comes from sampling the data, not from the algorithm. Regularization and model selection address it. For the rest of the chapter we therefore set generalization aside and take the objective at face value. Even so, the problem is hard. A deep network's objective has no closed-form minimizer like the one we found

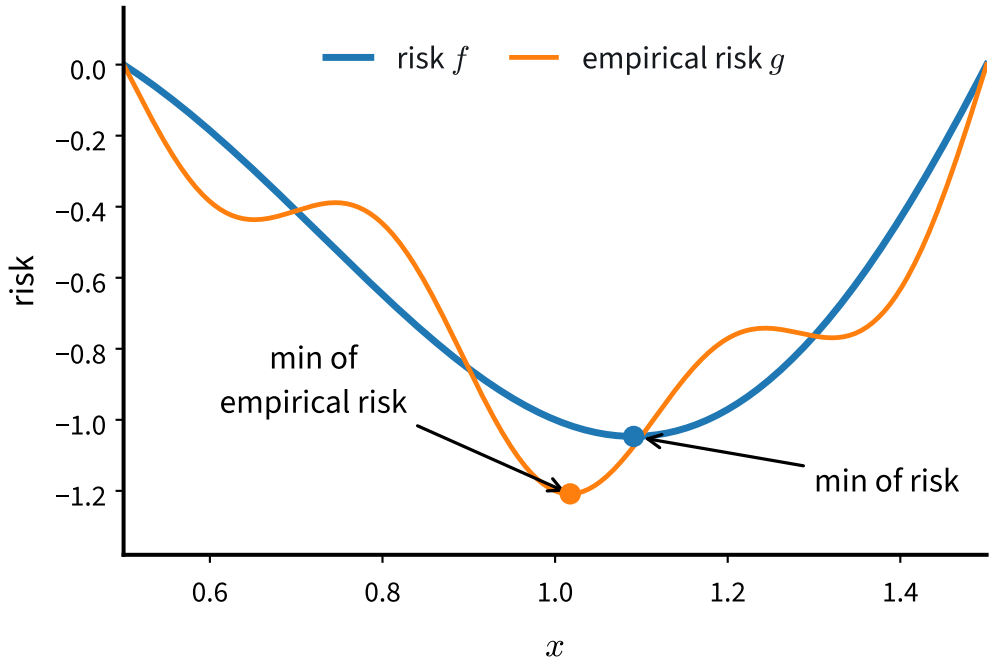


Figure 9.1: The optimizer minimizes the empirical risk g, which wobbles around the risk f; their minima lie in different places.

for linear regression in Section 2.1, so every algorithm in this chapter is iterative, and the surface it iterates over is far from a convex bowl.

9.1.2 Critical Points and Plateaus

An iterative method needs a signal to follow, and the gradient is that signal. The signal gives out at *critical points*, where the gradient is zero, and nearly gives out on *plateaus*, where it is tiny. These are the classical hazards of nonconvex optimization, and we look at each in turn.

9.1.2.1 Local Minima

If the value of f at x is smaller than at any nearby point, then x is a *local minimum*. If it is smallest over the entire domain, x is the *global minimum*. The function

$$f(x) = x\cos(\pi x), \quad -1 \le x \le 2, \tag{9.1}$$

has one of each (Figure 9.2).

An iterate that lands near either minimum sees its gradient shrink toward zero, so from the gradient alone a local minimum cannot be told apart from the global one. Deep learning objectives have many local minima. Noise in the gradient can move the parameters out of a shallow basin. This is one reason, as we will see below, why minibatch noise is not purely a nuisance.

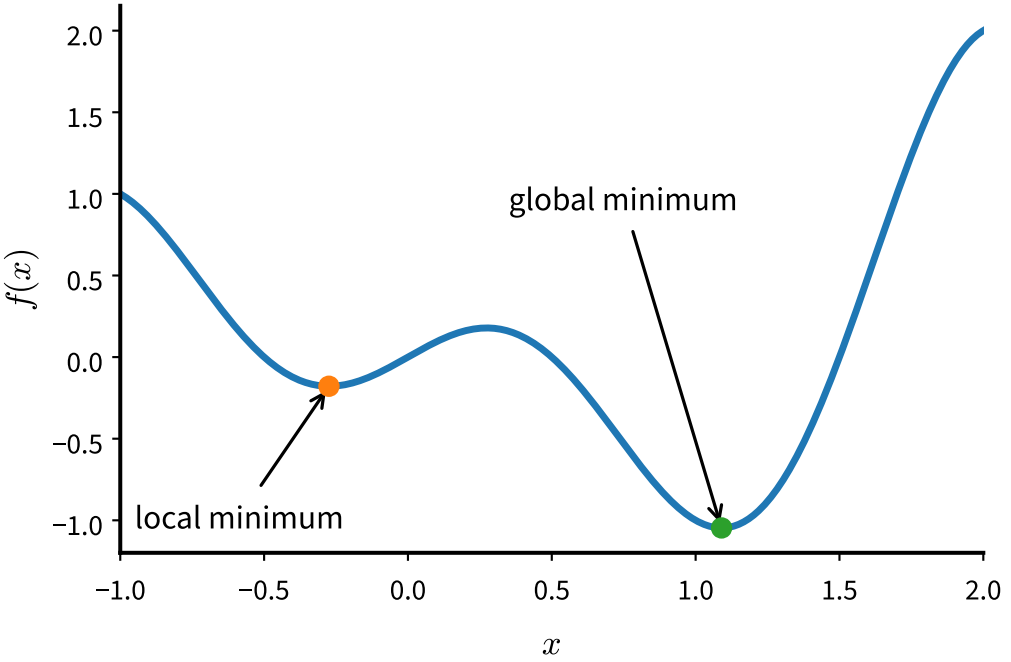


Figure 9.2: $f(x) = x\cos(\pi x)$ has a local minimum that is not global; near either the gradient vanishes.

9.1.2.2 Saddle Points

A *saddle point* is a critical point that is neither a minimum nor a maximum. In one dimension the closest example is $f(x) = x^3$ (Figure 9.3): its first and second derivatives both vanish at $x = 0$, so gradient descent can stall there, although the function keeps decreasing to the left. Strictly speaking, this point is a *stationary inflection*. A true saddle, which curves up in one direction and down in another, needs at least two dimensions; we use "saddle" for both.

The function $f(x, y) = x^2 - y^2$ has a true saddle at the origin (Figure 9.4). Along the x-axis the origin is a minimum, along the y-axis it is a maximum, and the surface has the shape of the saddle that gives the point its name.

In high dimensions saddles become the rule. To see why, recall the second-order test for a critical point of a function of k variables. Its Hessian, the $k \times k$ matrix of second derivatives, has k real eigenvalues, one curvature per principal direction, and their signs decide the shape:

- all k eigenvalues positive (the Hessian is *positive definite*): a strict local minimum;
- all k negative: a strict local maximum;
- some positive and some negative (the Hessian is *indefinite*): a saddle point, called *strict* because at least one direction curves downward;
- no negative and some zero eigenvalues, or no positive and some zero: the test is inconclusive, and higher-order terms decide.

A strict local minimum needs all k curvatures to be positive. If the signs were independent

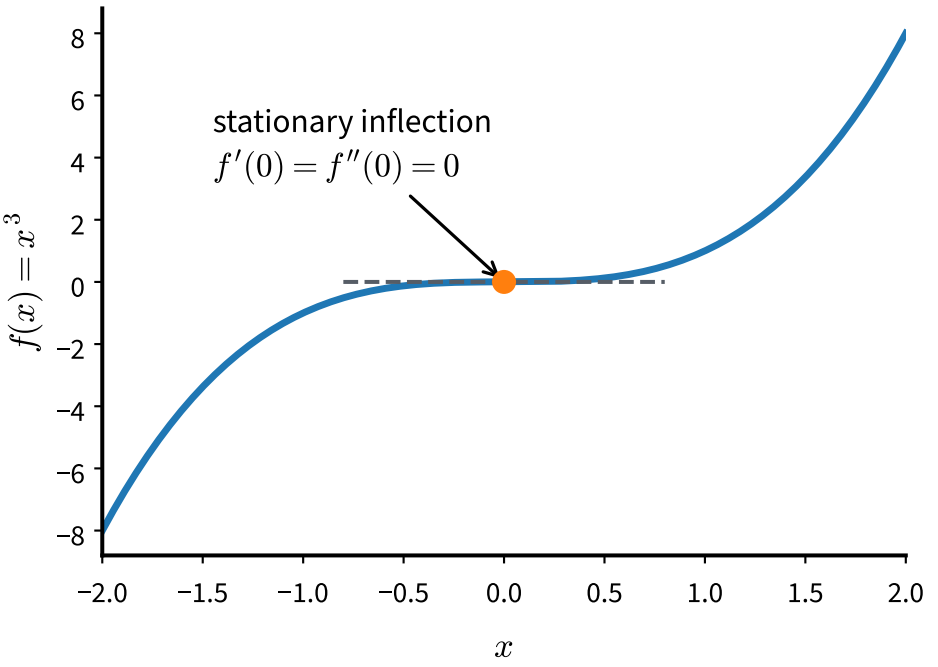


Figure 9.3: At $x = 0$ the cubic $f(x) = x^3$ has $f' = f'' = 0$ yet no extremum: a stationary inflection.

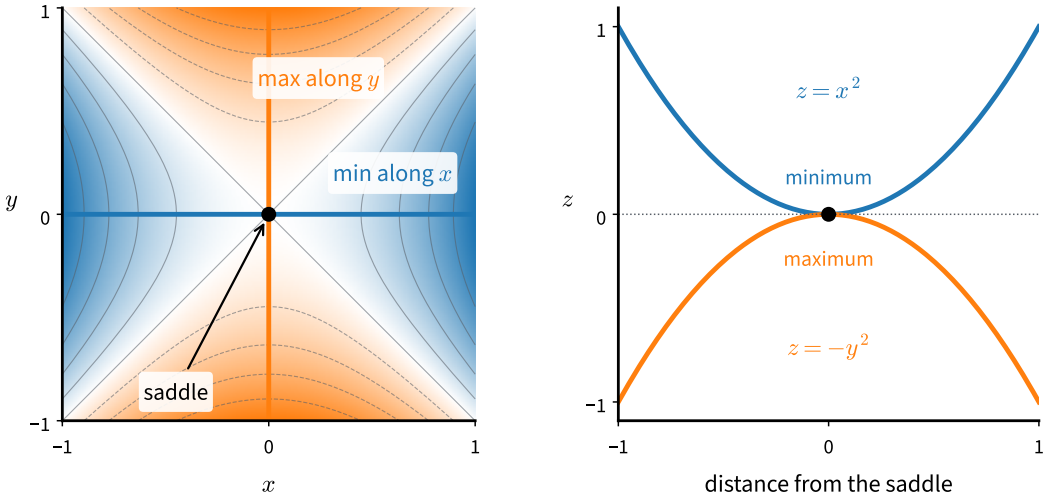


Figure 9.4: The saddle $z = x^2 - y^2$: a minimum along x and a maximum along y (left), seen in the two slices through the origin (right).

fair coin flips, a critical point of a function with $k = 1000$ parameters would be a minimum with probability $2^{-1000} \approx 10^{-301}$, and nearly every critical point would be a saddle. The coin-flip model is only a heuristic. The eigenvalues of a Hessian are not independent, as Exercise 6 shows. At the critical points of real networks, the fraction of negative eigenvalues falls as the loss falls: high-loss critical points are mostly saddles, while low-loss ones are close to minima (Dauphin et al. 2014). The measurements agree with the heuristic where it matters: in high dimensions the critical points a method meets on its way down are overwhelmingly saddles, not poor local minima.

9.1.2.3 Plateaus

A gradient can also be too small to guide an update at a point that is not critical at all. Suppose we minimize $f(x) = \tanh(x)$, starting from $x = 4$. The derivative is $f'(x) = 1 - \tanh^2(x)$, so $f'(4) \approx 0.0013$. The surface is nearly flat at the starting point (Figure 9.5), and gradient descent barely moves for a long time before it reaches the slope.

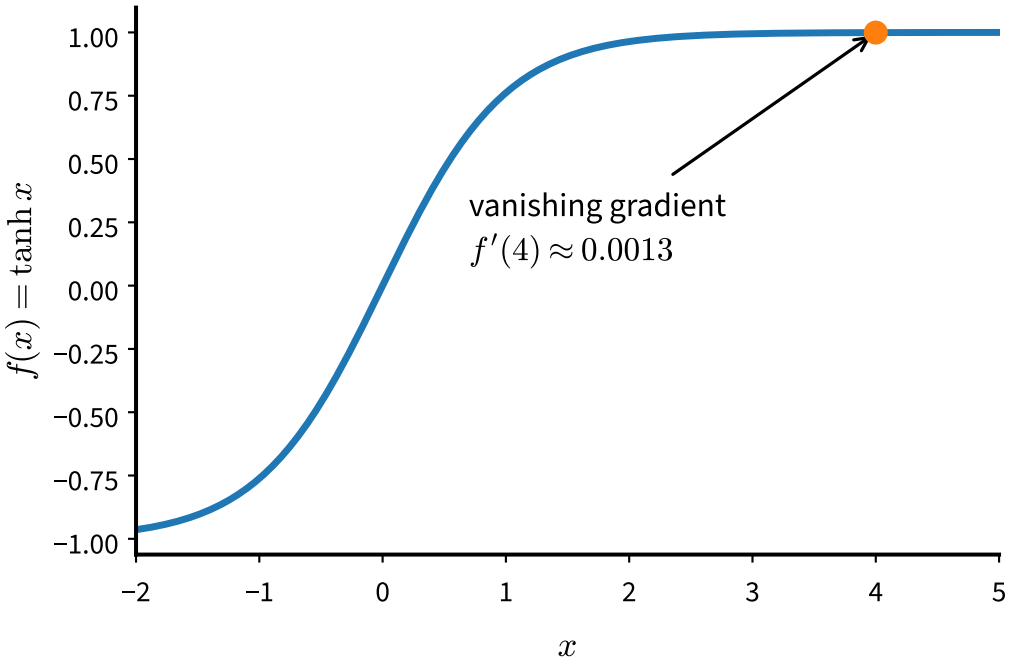


Figure 9.5: Minimizing $f(x) = \tanh x$ from $x = 4$ stalls: the surface is nearly flat, slope about 0.0013, a vanishing gradient with no critical point.

This is the vanishing-gradient problem of Section 4.4 in one variable. In a deep network the gradient of an early layer is a product of one derivative per later layer, and a product of many factors below one shrinks fast. Saturating activations made deep networks hard to train until ReLU activations and careful initialization removed most such plateaus. Those fixes belong to model design (Section 4.1.4), not to the optimizer.

The classical hazards are real, but two facts soften them. Deep learning does not need *the* global minimum: a good local minimum serves as well, and at low loss most critical points are good. And what limits the speed of routine training is usually something else, which we

turn to next.

9.1.3 Curvature and Noise

Two effects make training slow or unstable far more often than critical points do. The curvature of the objective differs from direction to direction, and the gradient we compute is only an estimate. Most methods in this chapter address one of these effects or both.

9.1.3.1 An Ill-Conditioned Valley

The smallest example of uneven curvature is a quadratic valley in two variables,

$$f(\mathbf{x}) = 0.1x_1^2 + 2x_2^2. \tag{9.2}$$

It curves gently along x_1, with second derivative $h_1 = 0.2$, and steeply along x_2, with second derivative $h_2 = 4$, so its contours are long, narrow ellipses. The gradient is $(0.2x_1, 4x_2)$. Gradient descent, the simplest optimizer, steps against the gradient with a learning rate η: $\mathbf{x} \leftarrow \mathbf{x} - \eta\nabla f(\mathbf{x})$. On this valley each step multiplies the two coordinates by fixed factors,

$$x_1 \leftarrow (1 - 0.2\eta)\, x_1, \qquad x_2 \leftarrow (1 - 4\eta)\, x_2.$$

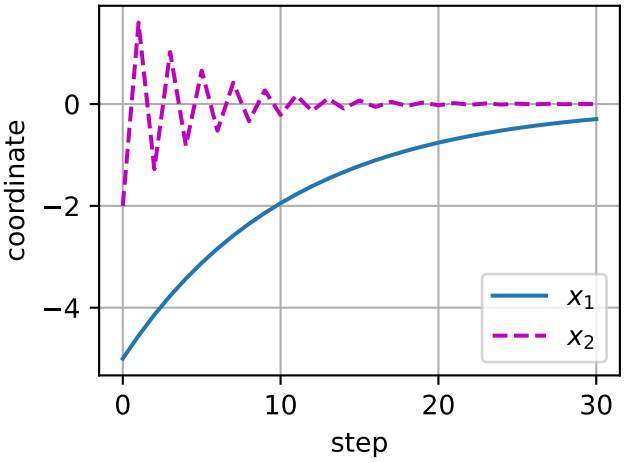
The steep coordinate sets a ceiling: x_2 shrinks only if $|1 - 4\eta| < 1$, that is, if $\eta < 0.5$. The flat coordinate sets the pace: below that ceiling, $1 - 0.2\eta$ stays above 0.9, so each step removes less than a tenth of the distance along x_1. This valley is the running example of the chapter: gradient descent, stochastic gradient descent, momentum, and Adam are all tested on it.

We run 30 steps from $(-5, -2)$ at $\eta = 0.45$, just under the ceiling, and plot both coordinates against the step count.

```
h = np.array([0.2, 4.0])  # Curvatures h1, h2 along x1 and x2

def gd_valley(eta, steps, x0=(-5.0, -2.0)):
    """Gradient descent on the valley; returns all iterates."""
    xs = [np.array(x0)]
    for _ in range(steps):
        xs.append(xs[-1] - eta * h * xs[-1])  # The gradient is h * x
    return np.stack(xs)

xs = gd_valley(0.45, steps=30)
d2l.plot(np.arange(31), [xs[:, 0], xs[:, 1]], 'step', 'coordinate',
         legend=['$x_1$', '$x_2$'])
```



The steep coordinate overshoots the valley floor on every step. Its factor is $1 - 4 \cdot 0.45 = -0.8$, so x_2 flips sign and loses a fifth of its size each time. The flat coordinate never overshoots, but its factor is 0.91, so it sheds only nine percent of its remaining distance per step. At that rate every factor of ten along x_1 takes about 24 steps, and after 30 steps x_1 is still -0.3. In the plane, the iterate zigzags across the narrow valley while it creeps along its floor.

Could another learning rate do better? We count the steps until both coordinates are within 1% of the minimum, measured relative to where they started.

```
print('  eta   factor x1   factor x2   steps to 1%')
for eta in [0.1, 0.2, 0.3, 0.4, 0.45, 2 / 4.2, 0.49, 0.5]:
    xs = gd_valley(eta, steps=1000)
    done = np.all(np.abs(xs) <= 0.01 * np.abs(xs[0]), axis=1)
    steps = int(np.argmax(done)) if done.any() else 'never'
    print(f'{eta:5.3f}   {1 - 0.2 * eta:9.3f}   {1 - 4 * eta:9.3f}   '
          f'{steps:>11}')
```

```
  eta   factor x1   factor x2   steps to 1%
0.100       0.980       0.600           228
0.200       0.960       0.200           113
0.300       0.940      -0.200            75
0.400       0.920      -0.600            56
0.450       0.910      -0.800            49
0.476       0.905      -0.905            47
0.490       0.902      -0.960           113
0.500       0.900      -1.000         never
```

Small learning rates are slow because the flat factor stays near one. Raising η helps until the steep factor, which grows more negative, overtakes the flat one in size. The best rate balances the two: $1 - 0.2\eta = -(1 - 4\eta)$ gives $\eta = 2/4.2 \approx 0.476$, where both factors have size 0.905, and the count bottoms out at 47 steps. Just above it the steep coordinate becomes the slow one, and at $\eta = 0.5$ it stops shrinking altogether. No choice of η reaches 1% in fewer than 47 steps.

The quantity behind this floor is the ratio of the largest to the smallest curvature, the *condi-*

tion number

$$\kappa = \frac{\lambda_{\max}}{\lambda_{\min}}, \tag{9.3}$$

where $\lambda_{\max}$ and $\lambda_{\min}$ are the largest and smallest eigenvalues of the Hessian. The valley's Hessian is diagonal, so its eigenvalues are the two curvatures, $\lambda_{\min} = h_1 = 0.2$ and $\lambda_{\max} = h_2 = 4$, and $\kappa = 20$. At the best rate, $\eta = 2/(\lambda_{\min} + \lambda_{\max})$, both factors have size $(\kappa - 1)/(\kappa + 1) \approx 1 - 2/\kappa$, which is $19/21 = 0.905$ for the valley. Reducing the error by a factor ϵ then takes about $(\kappa/2)\log(1/\epsilon)$ steps, and for the valley this is $10 \log 100 \approx 46$, the count we measured. Section 35.1.2 derives the formula. The cost of gradient descent grows linearly with κ, and the κ of a deep network is far larger than 20. The Hessian of a trained network has a few large eigenvalues and a bulk of eigenvalues near zero (Ghorbani et al. 2019), so its largest curvature exceeds the typical one by factors of hundreds or more.

9.1.3.2 The Edge of Stability

The valley treats curvature as a fixed property of the surface, and the classical advice follows from it: measure the largest curvature $\lambda_{\max}$, called the *sharpness*, and choose $\eta < 2/\lambda_{\max}$. On real networks the dependence can run the other way. Cohen et al. (2021) trained networks with full-batch gradient descent and found that the sharpness rises during training, a behavior they named *progressive sharpening*, until it reaches about $2/\eta$, and then stays there. The loss keeps falling, although not on every step, in a regime that the quadratic analysis calls unstable. They called this regime the *edge of stability*.

We can watch it happen on a network small enough to compute its full Hessian. The network has one input, eight tanh hidden units, and one output, 25 parameters in all, and it fits 16 points of $\sin(3x)$ on $[-2, 2]$ with the squared loss. We train it from one random start with full-batch gradient descent at $\eta = 0.4$ and at $\eta = 0.25$, whose ceilings $2/\eta$ are 5 and 8, and compute the sharpness every 500 steps.

```
X = torch.linspace(-2, 2, 16)
Y = torch.sin(3 * X)

def mlp_loss(p):  # 8 hidden units: weights, biases, output weights, bias
    W1, b1, W2, b2 = p[:8], p[8:16], p[16:24], p[24]
    return 0.5 * ((torch.tanh(X[:, None] * W1 + b1) @ W2 + b2 - Y) ** 2).mean()

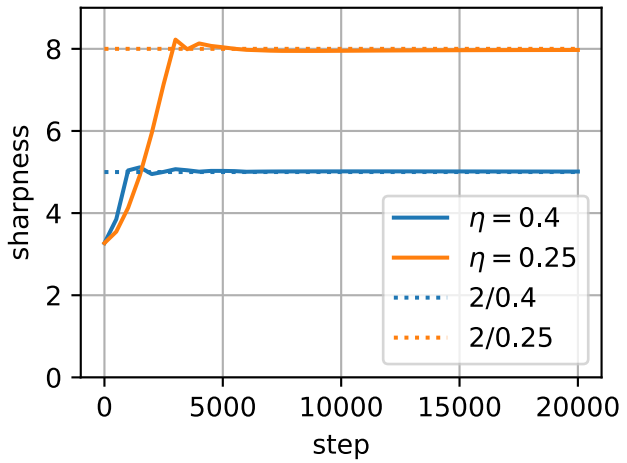
def sharpness(p):  # Largest eigenvalue of the Hessian
    return torch.linalg.eigvalsh(torch.func.hessian(mlp_loss)(p))[-1]

def gd_sharpness(eta, p, steps=20000, every=500):
    """Full-batch gradient descent; returns the sharpness every `every`
    steps and the final loss."""
    grad, sharp = torch.func.grad(mlp_loss), []
    for k in range(steps + 1):
        if k % every == 0:
            sharp.append(sharpness(p).item())
        p = p - eta * grad(p)
    return sharp, mlp_loss(p).item()
```

```
p0 = 0.5 * torch.randn(25)  # One random start for both runs
```

```
runs = {eta: gd_sharpness(eta, p0) for eta in (0.4, 0.25)}
for eta, (sharp, loss) in runs.items():
    print(f'eta {eta}: sharpness {float(sharp[0]):.2f} -> '
          f'{float(sharp[-1]):.2f} (2/eta = {2 / eta:.0f}), loss '
          f'{float(mlp_loss(p0)):.3f}'
          f' -> {float(loss):.1e}')
steps = np.arange(0, 20001, 500)
ceilings = [np.full(len(steps), 2 / eta) for eta in (0.4, 0.25)]
d2l.plot(steps, [runs[0.4][0], runs[0.25][0], *ceilings],
         'step', 'sharpness', ylim=(0, 9), fmts=('C0-', 'C1-', 'C0:', 'C1:'),
         legend=[r'$\eta = 0.4$', r'$\eta = 0.25$', '$2/0.4$', '$2/0.25$'])
```

```
eta 0.4: sharpness 3.27 -> 5.01 (2/eta = 5), loss 1.171 -> 4.0e-04
eta 0.25: sharpness 3.27 -> 7.97 (2/eta = 8), loss 1.171 -> 2.3e-04
```



Both runs start from the same weights, with a sharpness well below 8 and usually below 5. Neither stays there. In almost every run the sharpness climbs until it reaches the ceiling of its own learning rate and then stays close to $2/\eta$: near 5 for $\eta = 0.4$ and near 8 for $\eta = 0.25$. Meanwhile the loss keeps falling by orders of magnitude, though not on every step. The two runs differ only in η, yet they end in regions of different curvature, each as sharp as its own learning rate allows.

Gradient descent stays at the edge because it corrects itself. When the sharpness exceeds $2/\eta$, the iterate starts to oscillate along the sharpest direction, as x_2 did on the valley, and this oscillation moves the iterate toward flatter parts of the surface until the sharpness drops back below the ceiling (Damian et al. 2023). The quadratic model predicts the oscillation but not what follows. The third-order term of the Taylor expansion, which the quadratic model drops, turns the oscillation into a drift toward lower sharpness. The quadratic analysis misses this change of the surface under the iterate. Most results quoted in this chapter assume monotone descent on a fixed surface, and the edge of stability lies outside that assumption.

The start of training is delicate for a related reason. In large networks the sharpness is often

high at initialization and falls early in training, the reverse of our small network. A learning rate that suits the later surface can then diverge on the first steps, and warmup keeps η small until the sharpness has fallen (Kalra and Barkeshli 2024) (Section 9.6).

9.1.3.3 Noisy Gradients

The second effect is that the gradient we use is an estimate. The objective is an average of n per-example losses, so its exact gradient costs a pass over all n examples, far too much for a single step on a large dataset. Practical methods compute the gradient on a *minibatch* of b examples instead. This estimate is correct on average, and because it averages b independent terms, its variance is $1/b$ times the variance of a single example's gradient.

Noise changes what a constant learning rate can achieve. Near the minimum the true gradient is small, but the noise in its estimate is not, so the iterates never settle. They keep moving inside a region around the minimum, a *noise ball*, whose size grows with η. Convergence therefore needs a learning rate that decays over time, and the batch size b becomes a second dial for the noise, with its own computational cost.

Noise is not only a nuisance. It can push the iterate off a saddle point or out of a shallow basin. Gradient descent with a small enough learning rate escapes strict saddles from a random start even without noise, because the starting points that lead into a strict saddle form a set of measure zero (Lee et al. 2016). Escape can still take many steps, and noise shortens it. Noise rarely lifts the iterate out of a deep basin.

9.1.4 The Role of Convexity

Every surface in this section except the valley has been nonconvex. Yet the words we used to describe them (condition number, convergence rate, noise ball) come from *convex* analysis, where they are the subjects of theorems. A convex function has no saddle points and no local minima other than its global ones, so any weakness an optimizer shows on a convex problem belongs to the optimizer. A method that fails on a quadratic is unlikely to succeed on a transformer, which is why the first methods of this chapter meet the valley before they meet a network.

Convex analysis also applies locally. Near a minimum, a smooth loss is approximately a quadratic bowl, so the valley describes the last phase of training reasonably well even when the surface as a whole is far from convex. Some practical methods come directly from this local approximation: averaging the iterates near the bottom of the bowl, as in stochastic weight averaging (Izmailov et al. 2018), is a convex-analysis idea that works on deep networks (Section 9.10).

The local approximation does not extend to the whole surface. Permuting the hidden units of a layer leaves the function a network computes unchanged, so every minimum comes with a large family of copies of itself, one per permutation. If the loss were convex, the average of these copies would be a minimizer too. That average is a network whose units in the layer are all identical. It is no more expressive than a network with one unit and usually fits worse, as Exercise 5 makes precise. For deep learning, convexity is a local approximation and a source of tools, never a global fact. The appendix develops convex sets and functions, Jensen's inequality, and duality (Section 35.3), and proves the convergence rates this chapter quotes (Chapter 35).

9.1.5 Three Decisions

What we have seen suggests that any optimizer can be described by three decisions. The first is the *descent direction*. The negative gradient is the steepest direction when the size of a step is measured by its Euclidean length, and on the valley it points mostly across the valley rather than along it. The second is the *step size over time*: how far to move along the chosen direction, limited by the largest curvature, and how that distance should change as training proceeds. The third is the *treatment of noise*: the batch size and any averaging of gradients over steps decide how noisy each update is.

Each method in this chapter is a particular set of these decisions. Gradient descent (Section 9.2) makes the plainest choice for each and shows how dividing by the curvature would remove the condition number of the valley altogether. Stochastic gradient descent (Section 9.3) brings in the noise that the valley lacks. Momentum (Section 9.4) averages gradients over steps, which builds up speed along the flat floor of the valley and also damps the noise. Adam (Section 9.5) rescales the step coordinate by coordinate, and Muon (Section 9.7) measures the length of a step differently and so changes the direction itself. Schedules and the batch size settle the remaining two decisions in the later sections.

9.1.6 Summary

Optimization minimizes the empirical risk, while learning wants a low risk on new data, and no optimizer can close the gap between the two. On the training objective, gradients vanish at local minima, at saddle points, which dominate in high dimensions, and on plateaus such as saturated activations. Routine training is limited more often by two other effects. Uneven curvature forces one learning rate to serve both steep and flat directions, so gradient descent needs a number of steps proportional to the condition number κ: on the valley, with $\kappa = 20$, no learning rate reaches 1% in fewer than 47 steps. Noise in minibatch gradients keeps a constant learning rate from converging. Real networks also adapt their curvature to the step size and train at the edge of stability. The rest of the chapter builds optimizers that choose direction, step size, and noise level better than plain gradient descent.

9.1.7 Exercises

1. **Critical points and plateaus.** Gradient descent slows or stops at each of the following points: the critical points of $f(x) = x\cos(\pi x)$ in the interior of $[-1, 2]$, the origin of $f(x) = x^3$, the origin of $f(x, y) = x^2 - y^2$, and $x = 4$ for $f(x) = \tanh x$.
    1. Compute the gradient and the Hessian at each point, locating the critical points of $x\cos(\pi x)$ numerically. Classify each point by the eigenvalue test of Section 9.1.2.2 as a local minimum, local maximum, saddle, inconclusive case, or non-critical point.
    2. Gradient descent with learning rate η on $f(x) = x^3$ starts at $x_0 = \epsilon$ or at $x_0 = -\epsilon$ for a small $\epsilon > 0$. For each sign, determine whether the iterates approach 0 or move away, and find the rate at which x_t decays on the side that approaches. Explain why the escape of strict saddles stated in Section 9.1.3.3 does not apply to this point.
    3. Using $1 - \tanh^2 x \approx 4e^{-2x}$ for large x, estimate how many steps gradient descent at $\eta = 1$ needs to move from $x = 4$ to $x = 2$ on $f(x) = \tanh x$.

    Adapted from Simon Prince, Understanding Deep Learning, Problem 6.6.

2. code **Saddle escape time.** Balancing a ball on the saddle $f(x_1, x_2) = x_1^2 - x_2^2$ of Figure 9.4 is stable along x_1 and unstable along x_2. Gradient descent with learning rate η started at $(1, \delta)$ behaves the same way.
    1. Write the iterates in closed form and derive the number of steps until $|x_2| \geq 1$ as a function of δ and η. Evaluate it for $\delta \in \{10^{-2}, 10^{-6}\}$ at $\eta = 0.1$, check both counts numerically, and state what happens for $\delta = 0$.
    2. Add zero-mean Gaussian noise of standard deviation σ to every gradient evaluation and measure the escape time from $\delta = 0$ for $\sigma \in \{10^{-3}, 10^{-1}\}$. Reconcile the result with the statement in Section 9.1.3.3 that gradient descent from a random start escapes strict saddles without noise.

3. code **An ill-conditioned valley.** Gradient descent on the valley Equation 9.2 multiplies x_1 by $1 - 0.2\eta$ and x_2 by $1 - 4\eta$ on every step, so it is stable for $\eta < 0.5$. The *step count* is the number of steps until both $|x_1|$ and $|x_2|$ are below 1% of their starting values.
    1. For $f(\mathbf{x}) = \frac{\lambda_{\min}}{2} x_1^2 + \frac{\lambda_{\max}}{2} x_2^2$, set $\eta = 1.8/\lambda_{\max}$, the same fraction of the stability limit $2/\lambda_{\max}$ as $\eta = 0.45$ is of 0.5. Predict the step count from the two factors for $\kappa = \lambda_{\max}/\lambda_{\min} \in \{10, 100, 1000\}$, check it numerically, and compare with the linear growth in κ that Section 35.1.2 derives.
    2. Derive the best rate that the section quotes: show that the learning rate minimizing the larger of the two factors $|1 - \eta\lambda_{\min}|$ and $|1 - \eta\lambda_{\max}|$ is $\eta^\star = 2/(\lambda_{\min} + \lambda_{\max})$, and that the factor it achieves is $(\kappa - 1)/(\kappa + 1)$. Use the result to explain why tuning the learning rate cannot remove the dependence of the step count on κ, and compare your derivation with the appendix's Equation 35.9.

4. code **Learning rate and sharpness.** A two-layer scalar linear network $\hat{y} = abx$ is fit to the single example $(x, y) = (1, 1)$ with the loss $\ell(a, b) = \frac{1}{2}(ab - 1)^2$. Every point of the curve $ab = 1$ is a minimizer.
    1. Show that the Hessian at a minimizer (a, b) has largest eigenvalue $\lambda_{\max} = a^2 + b^2$. Which minimizers are flattest?
    2. Run 200 steps of gradient descent from $(a, b) = (4, 0.1)$ at $\eta \in \{0.05, 0.1, 0.15, 0.2\}$. For each run, report the $\lambda_{\max}$ of the minimizer it reaches next to $2/\eta$.
    3. Explain the pattern with the stability condition $\eta < 2/\lambda_{\max}$.

5. **Permutation-symmetric minima.** ▯ Consider a multilayer perceptron with one hidden layer of $d \geq 2$ units and a single output, and let $\theta^\star$ be a local minimum of its loss at which no two hidden units have identical incoming weights.
    1. Show that permuting the hidden units yields $d!$ distinct parameter vectors that all compute the same function and are therefore all local minima with the same loss.
    2. Suppose the loss were a convex function of the parameters. Show that the average of the $d!$ copies would then also be a global minimizer and that it computes the same function as a network with one hidden unit. Conclude that the loss is not convex whenever every network with one hidden unit has a higher loss than $\theta^\star$.
    3. For the activation tanh, identify a second symmetry of each hidden unit, state the resulting count of equivalent minima, and give a condition on $\theta^\star$ under which all of them are distinct.

6. **Eigenvalues of symmetric random matrices.** ▯ Let $\mathbf{M}$ be a random symmetric $k \times k$

matrix whose entries $M_{ij} = M_{ji}$, $i \le j$, are drawn independently from distributions p_{ij} with $p_{ij}(x) = p_{ij}(-x)$ (Wigner 1958).

1. Prove that an eigenvalue λ drawn uniformly at random from the k eigenvalues of $\mathbf{M}$ satisfies $P(\lambda > 0) = P(\lambda < 0)$.
2. Explain why this symmetry alone does not determine the fraction of negative eigenvalues in a single realization of $\mathbf{M}$. State what the semicircle law of Wigner (1958) says about that fraction for large k when the off-diagonal entries are identically distributed with finite variance.
3. The coin-flip heuristic of Section 9.1.2.2 treats the k signs as independent, so all of them are positive with probability 2^{-k}. For $k = 2$, draw the diagonal entries from $\mathcal{N}(0, 1)$ and the off-diagonal entry from $\mathcal{N}(0, 1/2)$. Compute the probability that both eigenvalues are positive, compare it with 2^{-2}, and state what the gap says about the independence assumption.

9.2 Gradient Descent

Every model in this book was trained with a learning rate fixed before the run, usually copied from a recipe and lowered if the loss diverged. The previous section showed why the choice is delicate: on a valley that curves steeply in one direction and gently in another, a step small enough for the steep direction makes little progress along the gentle one. This section asks what decides which learning rates work, and how fast the best of them can go. We answer it for gradient descent, the simplest optimizer and the ancestor of every other method in this chapter: compute the gradient, take a step against it, and repeat.

Gradient descent has a single knob, the learning rate η, and on the valley $f(\mathbf{x}) = 0.1x_1^2 + 2x_2^2$ of Section 9.1 this knob already decides everything. At $\eta = 0.45$ the iterate needs 49 steps to come within 1% of the minimum, at $\eta = 0.5$ it never arrives, and no learning rate does better than 47 steps. A learning rate that makes a large network diverge does so for the same reason that it makes a one-dimensional quadratic diverge, so the small cases are worth understanding completely.

We split the question into three. Why does a step against the gradient lower the objective, and how large may the step be? What would the ideal step be if we also knew the curvature? And how can we approximate that ideal cheaply? The answers are a Taylor expansion, Newton's method, and preconditioning. Preconditioning is the idea behind the adaptive methods of Section 9.5. Throughout the section the gradient is exact; Section 9.3 replaces it with a noisy estimate.

9.2.1 One-Dimensional Gradient Descent

Why should a step against the gradient reduce the objective at all? The one-dimensional case already contains the answer. Let $f : \mathbb{R} \to \mathbb{R}$ be continuously differentiable. For a small change ϵ in its argument, its Taylor expansion reads

$$f(x + \epsilon) = f(x) + \epsilon f'(x) + O(\epsilon^2). \tag{9.4}$$

Near x the function is close to its tangent line, so a move downhill along the tangent should decrease f. We pick a step size $\eta > 0$, set $\epsilon = -\eta f'(x)$, and substitute:

$$f(x - \eta f'(x)) = f(x) - \eta f'(x)^2 + O(\eta^2 f'(x)^2). \tag{9.5}$$

The first-order change $-\eta f'(x)^2$ is negative whenever the derivative is not zero, and for small enough η it outweighs the remainder. Iterating the update

$$x \leftarrow x - \eta f'(x)$$

therefore decreases f as long as the step is small. How small depends on how fast the slope changes. If f' is L-Lipschitz, that is, if the slope changes by at most L per unit of x, then every learning rate with $0 < \eta < 2/L$ decreases f; Chapter 35 proves this.

The quadratic $f(x) = x^2$ shows all of this in closed form. Its derivative is $2x$, so one step maps x to $x - 2\eta x$:

$$x \leftarrow (1 - 2\eta)\, x. \tag{9.6}$$

On a quadratic, gradient descent multiplies the distance to the minimum by a fixed factor at every step. The iterates converge if and only if $|1 - 2\eta| < 1$, that is, $0 < \eta < 1$, which is the bound $2/L$ with $L = 2$. With $\eta = 0.2$ the factor is 0.6, and ten steps from $x = 10$ should leave $10 \cdot 0.6^{10} \approx 0.06$. We check this numerically.

```
def f(x):  # Objective function
    return x ** 2

def f_grad(x):  # Its derivative
    return 2 * x
```

```
def gd(eta, f_grad, steps=10):
    """Gradient descent from x = 10; returns all iterates."""
    x = 10.0
    results = [x]
    for _ in range(steps):
        x -= eta * f_grad(x)
        results.append(float(x))
    print(f'step {steps}, x: {float(x):f}')
    return results

results = gd(0.2, f_grad)
```

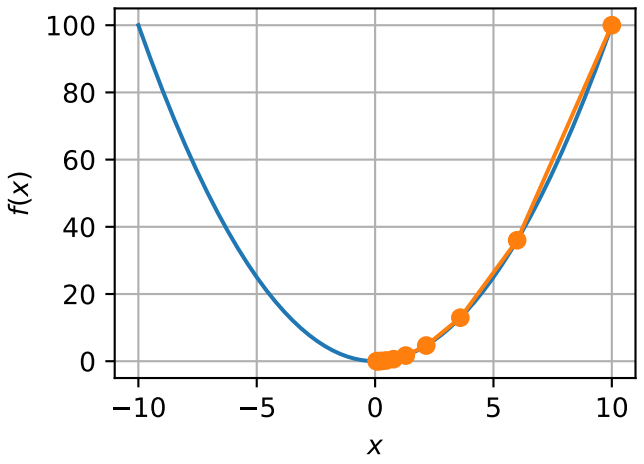
```
step 10, x: 0.060466
```

The iterates approach the minimum geometrically, each step keeping 60% of the distance. We plot them on the graph of f.

```
def show_trace(results, f):
    n = max(abs(min(results)), abs(max(results)))
    f_line = d2l.arange(-n, n, 0.01)
    d2l.set_figsize()
    d2l.plot([f_line, results], [f(f_line), [f(x) for x in results]],
```

```
                '$x$', '$f(x)$', fmts=['-', '-o'])

show_trace(results, f)
```

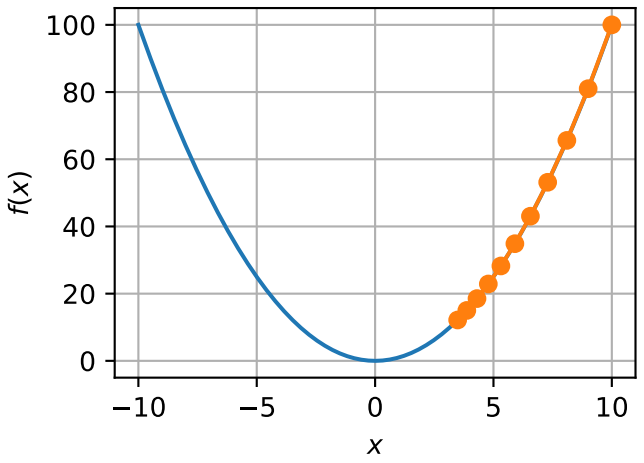


9.2.1.1 Learning Rate

The learning rate is ours to choose, and the choice can go wrong in either direction. With $\eta = 0.05$ the factor is 0.9, and ten steps leave $10 \cdot 0.9^{10} \approx 3.5$: the iterate barely moves, and reaching the minimum would take many more steps.

```
show_trace(gd(0.05, f_grad), f)
```

```
step 10, x: 3.486784
```

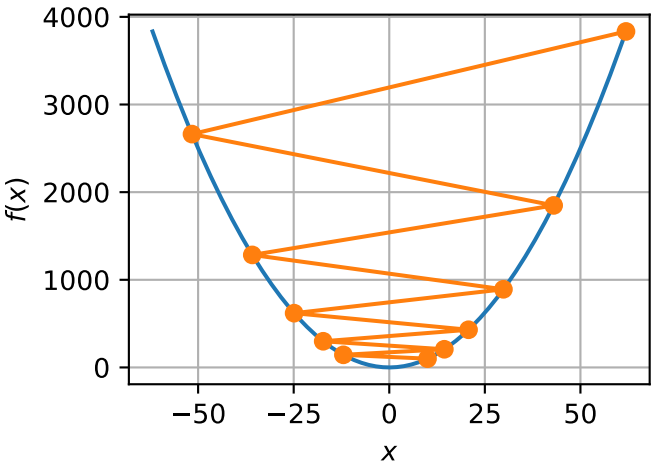


With $\eta = 1.1$ the factor is -1.2. Each step overshoots the minimum and lands farther away on the other side, so after ten steps the distance has grown to $10 \cdot 1.2^{10} \approx 62$. In the

language of Equation 9.5, the step $|\eta f'(x)|$ is too long for the tangent line to be a good guide, and the remainder term takes over.

```
show_trace(gd(1.1, f_grad), f)
```

```
step 10, x: 61.917364
```



The best learning rate for this quadratic is $\eta = 0.5$: the factor is zero, and a single step lands on the minimum. That rate is the inverse of the curvature $f''(x) = 2$, an observation that returns with Newton's method below.

9.2.1.2 Local Minima

On a convex parabola the only risk is the step size. A nonconvex function adds another: where the iterate ends up can depend on the learning rate. Consider $f(x) = x\cos(cx)$ with $c = 0.15\pi$. From $x = 10$ the deep basin around $x \approx 7.3$ is close by, and a shallow one lies near $x \approx -1.8$. We run gradient descent with $\eta = 0.5$ and with $\eta = 2$, and plot the second run.

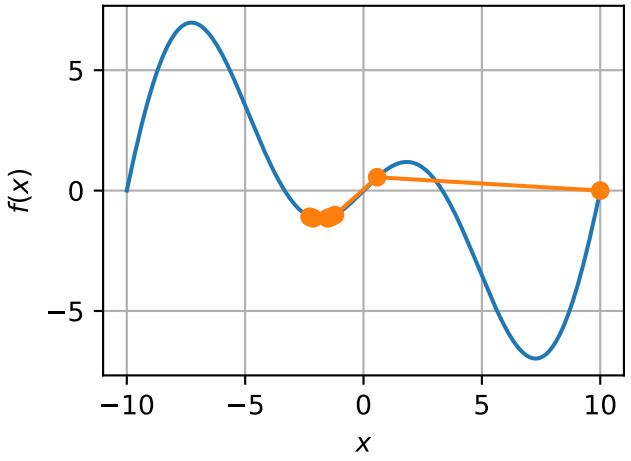
```
c = d2l.tensor(0.15 * np.pi)

def f_cos(x):  # Objective function
    return x * d2l.cos(c * x)

def f_cos_grad(x):  # Its derivative
    return d2l.cos(c * x) - c * x * d2l.sin(c * x)

gd(0.5, f_cos_grad)  # A moderate learning rate
show_trace(gd(2, f_cos_grad), f_cos)
```

```
step 10, x: 7.269388
step 10, x: -1.528166
```



With $\eta = 0.5$ the iterate rolls into the deep basin and stops near 7.27. With $\eta = 2$ the first step jumps over that basin to $x \approx 0.6$, and the iterate ends up bouncing around the shallow minimum. A learning rate this large is unrealistic for a one-dimensional problem, but large steps in the early phase of training play the same role in deep networks: they decide which region of the surface training explores.

9.2.2 Multivariate Gradient Descent

The same update works for a vector of parameters $\mathbf{x} = (x_1, \dots, x_d)$ and an objective $f : \mathbb{R}^d \to \mathbb{R}$. The gradient collects the d partial derivatives,

$$\nabla f(\mathbf{x}) = \left(\frac{\partial f(\mathbf{x})}{\partial x_1}, \dots, \frac{\partial f(\mathbf{x})}{\partial x_d} \right), \tag{9.7}$$

and each entry measures how fast f changes when its coordinate alone changes. The multivariate Taylor expansion has the same form as the one-dimensional one,

$$f(\mathbf{x} + \epsilon) = f(\mathbf{x}) + \epsilon^\top \nabla f(\mathbf{x}) + O(\|\epsilon\|^2). \tag{9.8}$$

Among all steps ϵ of a fixed small length r, which one decreases f the most? To first order the change is $\epsilon^\top \nabla f(\mathbf{x})$, and by the Cauchy–Schwarz inequality this inner product is at least $-r\|\nabla f(\mathbf{x})\|$, with equality exactly when ϵ points along $-\nabla f(\mathbf{x})$. The negative gradient is therefore the direction of *steepest descent*, and with a learning rate $\eta > 0$ we obtain gradient descent:

$$\mathbf{x} \leftarrow \mathbf{x} - \eta \nabla f(\mathbf{x}).$$

We return to the valley, $f(\mathbf{x}) = 0.1x_1^2 + 2x_2^2$, with curvatures $h_1 = 0.2$ along x_1 and $h_2 = 4$ along x_2. These are the eigenvalues $\lambda_{\min}$ and $\lambda_{\max}$ of its Hessian in Section 9.1, and the gradient is $(h_1 x_1, h_2 x_2) = (0.2x_1, 4x_2)$. The rest of the chapter returns to this valley, so we save it and its gradient.

```
VALLEY_CURVATURES = (0.2, 4.0)
h1, h2 = VALLEY_CURVATURES  # Curvatures of the valley along x1 and x2

def f_valley(x1, x2):
    """The ill-conditioned quadratic valley h1/2 x1^2 + h2/2 x2^2, with
    (h1, h2) = VALLEY_CURVATURES."""
    h1, h2 = VALLEY_CURVATURES
    return h1 / 2 * x1 ** 2 + h2 / 2 * x2 ** 2

def valley_grad(x1, x2):
    """Gradient of f_valley."""
    h1, h2 = VALLEY_CURVATURES
    return h1 * x1, h2 * x2
```

To follow the iterates in the plane we write a helper that the rest of the chapter reuses. `train_2d` starts at $(-5, -2)$ and applies an update rule with the gradient `f_grad` for a given number of steps. Besides the two coordinates and the gradient, the update rule receives and returns two state variables, which the methods with memory in later sections (momentum, Adam) use; gradient descent ignores them. Hyperparameters such as the learning rate are passed through as keyword arguments. The plotting utility `d2l.show_trace_2d` (the online appendix "Utility Functions and Classes") draws the path over the contour lines of the objective.

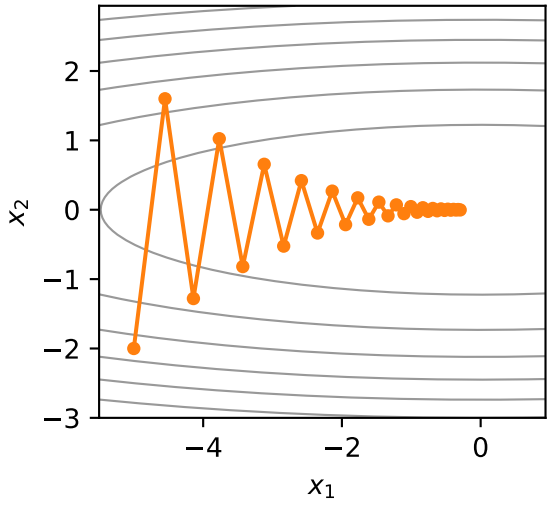
```
def train_2d(trainer, f_grad, steps=20, **hparams):
    """Run a 2D update rule from (-5, -2) and return the iterates."""
    x1, x2, s1, s2 = -5, -2, 0, 0  # s1, s2: state of stateful optimizers
    results = [(x1, x2)]
    for _ in range(steps):
        x1, x2, s1, s2 = trainer(x1, x2, s1, s2, f_grad, **hparams)
        results.append((x1, x2))
    print(f'step {steps}, x1: {float(x1):f}, x2: {float(x2):f}')
    return results
```

We run the same 30 steps at $\eta = 0.45$ as in Section 9.1, now drawn in the plane.

```
def gd_2d(x1, x2, s1, s2, f_grad, eta):
    """One step of gradient descent in 2D; the state stays zero."""
    g1, g2 = f_grad(x1, x2)
    return x1 - eta * g1, x2 - eta * g2, 0, 0

d2l.show_trace_2d(f_valley, train_2d(gd_2d, valley_grad, steps=30, eta=0.45))
```

```
step 30, x1: -0.295265, x2: -0.002476
```



Each step is perpendicular to the contour line through the current point, as steepest descent must be. On a narrow valley that direction points mostly across the valley: from $(-5, -2)$ the gradient is $(-1, -8)$, eight times steeper across than along. The iterate therefore zigzags from wall to wall and makes slow progress along the floor, and after 30 steps x_1 is still -0.3. A larger η would speed up progress along the floor but make the oscillation across the valley unstable. What the valley needs is a different learning rate in each direction.

9.2.3 Newton's Method

On the quadratic x^2 the best learning rate was the inverse of the curvature. The valley has two curvatures, 0.2 along x_1 and 4 along x_2, so a learning rate of $1/0.2 = 5$ along x_1 and $1/4$ along x_2 would solve each coordinate in a single step. Newton's method makes this idea general: it measures the curvature in every direction and divides the gradient by it.

The curvature enters through the second-order term of the Taylor expansion,

$$f(\mathbf{x} + \epsilon) = f(\mathbf{x}) + \epsilon^\top \nabla f(\mathbf{x}) + \frac{1}{2}\epsilon^\top \mathbf{H} \epsilon + O(\|\epsilon\|^3), \tag{9.9}$$

where $\mathbf{H} = \nabla^2 f(\mathbf{x})$ is the *Hessian*, the $d \times d$ matrix of second derivatives. Ignoring the remainder, the right-hand side is a quadratic in ϵ. Setting its gradient with respect to ϵ to zero (with the rules of Section 1.4.2) gives the step that minimizes it:

$$\nabla f(\mathbf{x}) + \mathbf{H}\epsilon = 0, \quad \text{hence} \quad \epsilon = -\mathbf{H}^{-1} \nabla f(\mathbf{x}).$$

Newton's method is gradient descent with the gradient multiplied by the inverse Hessian. Each direction receives the step its own curvature calls for, and there is no learning rate to tune. When f is itself a quadratic, the expansion is exact and a single Newton step lands on the minimum. On the valley, $\mathbf{H} = \text{diag}(0.2, 4)$ and $\mathbf{H}^{-1} \nabla f(\mathbf{x}) = \mathbf{x}$, so one step from any point reaches the origin, whatever the condition number.

On functions that are not quadratic, the method has to iterate. The convex function $f(x) = \cosh(cx)$ with $c = 0.5$ has its minimum at $x = 0$, and we run ten Newton steps from $x = 10$.

```
def f_cosh(x, c=d2l.tensor(0.5)):  # Objective function
    return d2l.cosh(c * x)

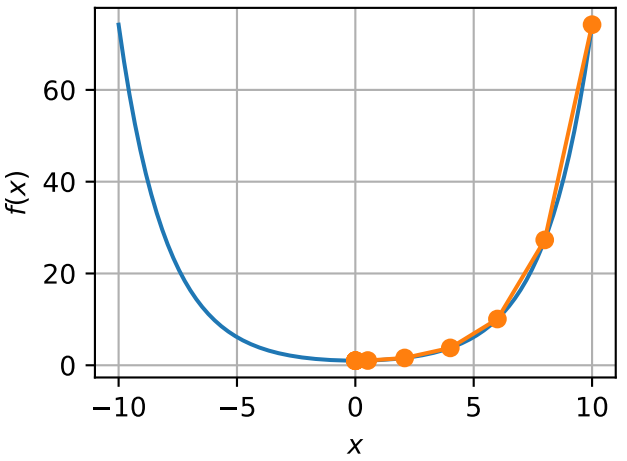
def f_cosh_grad(x, c=d2l.tensor(0.5)):  # First derivative
    return c * d2l.sinh(c * x)

def f_cosh_hess(x, c=d2l.tensor(0.5)):  # Second derivative
    return c**2 * d2l.cosh(c * x)

def newton(f_grad, f_hess, eta=1, steps=10):
    """Newton's method from x = 10, with an optional step size eta."""
    x = 10.0
    results = [x]
    for _ in range(steps):
        x -= eta * f_grad(x) / f_hess(x)
        results.append(float(x))
    print(f'step {steps}, x: {float(x):f}')
    return results

show_trace(newton(f_cosh_grad, f_cosh_hess), f_cosh)
```

```
step 10, x: 0.000000
```



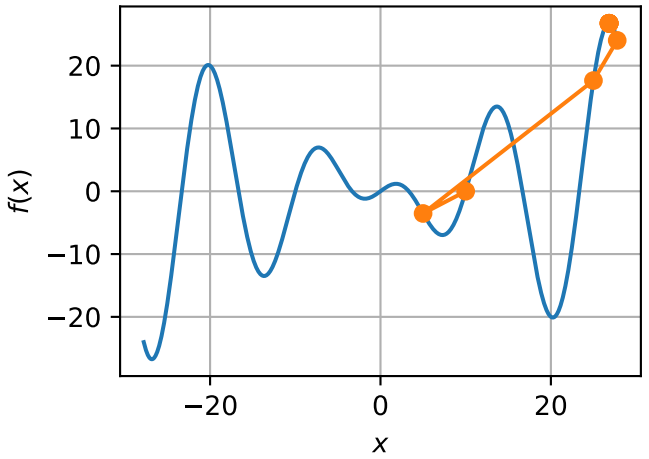
The iterate reaches the minimum to machine precision within ten steps, with no learning rate chosen by hand.

Newton's method divides by the curvature, and the curvature of a nonconvex function can be negative. Where it is, the step points uphill, toward a maximum. We return to $f(x) = x\cos(cx)$ with $c = 0.15\pi$, which has a minimum near 7.27, close to the start.

```
def f_cos_hess(x):  # Second derivative of f_cos
    return - 2 * c * d2l.sin(c * x) - x * c**2 * d2l.cos(c * x)

show_trace(newton(f_cos_grad, f_cos_hess), f_cos)
```

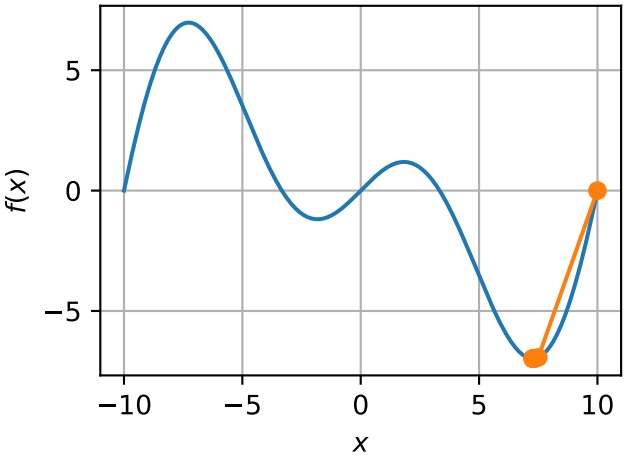
```
step 10, x: 26.834133
```



The first step overshoots to $x = 5$. The curvature there is positive but small, so the next step divides a large slope by a small number and jumps to $x \approx 25$, where the curvature is negative. From there the iteration climbs to the local maximum near 26.8, a point where the gradient vanishes as it does at a minimum. One remedy is to replace the Hessian by its absolute value, so that every step points downhill. Another is to bring back a learning rate. With $\eta = 0.5$ the first step stops at $x = 7.5$, inside the basin around the minimum, where the curvature is positive, and the iteration converges. It converges only linearly, however: each step halves the distance to 7.27. Damping makes the method safe here, and in exchange it gives up Newton's fast local rate.

```
show_trace(newton(f_cos_grad, f_cos_hess, eta=0.5), f_cos)
```

```
step 10, x: 7.269860
```



Damping is no guarantee: it only makes a jump into a region of negative curvature less likely. Two further facts about Newton's method matter here, and the appendix proves both. First, near a minimum with positive curvature it converges at least *quadratically*: the number of correct digits roughly doubles with every step (Section 35.2.1). Second, the method does not scale to deep learning.

Storing the Hessian of a model with d parameters takes $O(d^2)$ memory and solving a linear system with it takes $O(d^3)$ time. For $d = 10^9$ the Hessian has 10^{18} entries, four exabytes in single precision, before the first step. And a deep network's Hessian has negative eigenvalues, which steer the raw Newton step toward saddles and maxima. Classical optimization lowers these costs with curvature estimates built from successive gradients and with safer step rules (Section 35.2.2.2). Deep learning keeps a simpler part of the idea: rescale the gradient, direction by direction, with an inexpensive estimate of the curvature.

9.2.4 Preconditioning

A *preconditioner* rescales the gradient before the step. The simplest useful one keeps only the diagonal of the Hessian,

$$\mathbf{x} \leftarrow \mathbf{x} - \eta \,\mathrm{diag}(\mathbf{H})^{-1} \nabla f(\mathbf{x}), \tag{9.10}$$

which divides each coordinate of the gradient by the curvature along that coordinate. In effect, every coordinate receives its own learning rate. Storing the diagonal takes as much memory as storing the gradient.

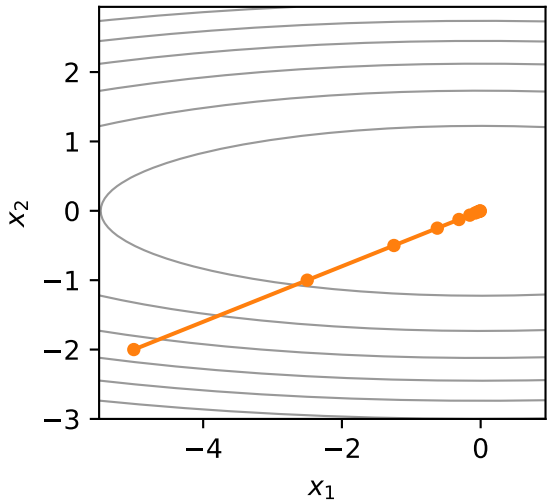
Ill-conditioning of this axis-aligned kind arises from something as mundane as units. Suppose a linear regression model receives two unrelated length features of similar size, one recorded in meters and the other in millimeters. The curvature of the squared loss along a weight is the mean of the squared feature, so the millimeter weight has a curvature 10^6 times larger than the meter weight. The units alone give a condition number of at least 10^6. A single learning rate small enough for the millimeter weight leaves the meter weight almost frozen. Dividing by the diagonal curvature removes the factor 10^6 without our ever noticing the units.

On the valley, the diagonal of the Hessian is $(0.2, 4)$, and we run the preconditioned update at $\eta = 0.5$.

```
def precond_gd_2d(x1, x2, s1, s2, f_grad, eta):
    g1, g2 = f_grad(x1, x2)
    return x1 - eta * g1 / h1, x2 - eta * g2 / h2, 0, 0  # diag(H) = (h1, h2)

d2l.show_trace_2d(f_valley, train_2d(precond_gd_2d, valley_grad, steps=10,
                                     eta=0.5))
```

```
step 10, x1: -0.004883, x2: -0.001953
```



Both coordinates now shrink by the same factor $1 - \eta = 0.5$ per step, so the path runs straight to the minimum. Seven steps bring both within 1%, against 47 for the best plain learning rate, and the count no longer depends on the condition number. At $\eta = 1$ one step suffices: because the Hessian of the valley is diagonal, the diagonal preconditioner is Newton's method.

Two limitations shape what comes next. A deep network does not give us its Hessian, not even the diagonal, so practical methods estimate a diagonal scale from the gradients they observe. AdaGrad, RMSProp, and Adam (Section 9.5) differ mainly in how they form that estimate. And a diagonal preconditioner only helps when the steep and flat directions line up with the coordinate axes. Exercise 2 rotates the valley and shows the preconditioner failing. For this reason later methods precondition whole weight matrices rather than single coordinates.

With an exact gradient, then, the step size and the curvature explain everything we have seen. But an exact gradient of a training loss requires a pass over the whole dataset. The next section takes up this problem.

9.2.5 Summary

Gradient descent decreases a differentiable function by stepping against the gradient, and the decrease is guaranteed only while the step is short enough for the first-order Taylor ex-

pansion to hold. On a quadratic each step multiplies the error by a fixed factor: a learning rate that is too small makes the factor close to one, and one that is too large makes it exceed one in size and the iterates diverge. On nonconvex functions the learning rate also decides which minimum the iterate reaches. Newton's method divides the gradient by the curvature and solves a quadratic in one step, whatever its condition number, but its cost and its attraction to maxima and saddles rule it out for deep networks. Preconditioning keeps the useful part: a diagonal rescaling solved the valley in seven steps instead of 47, and the adaptive methods of later sections estimate such a rescaling from gradients.

9.2.6 Exercises

1. code **Learning-rate sweep.** Extend `gd` to run at most 100 steps and to stop as soon as $|x| < 10^{-3}$. Run it on $f(x) = x^2$, $f(x) = |x|^{1.5}$, and $f(x) = x^4$ at $\eta \in \{0.01, 0.1, 0.5, 0.9, 1.1\}$.
    1. Tabulate, for each of the fifteen runs, the number of steps until the stopping test holds, or, for a run that misses the test within 100 steps, whether $|x|$ grew without bound or stayed bounded.
    2. The second derivative of $|x|^{1.5}$ is unbounded near the origin. Explain what the iterates do close to 0 at any fixed η, and why the stopping test is reached late or not at all.
    3. The second derivative of x^4 vanishes at the origin. Show that once the iterates are small, $x_{t+1} \approx x_t(1 - 4\eta x_t^2)$, and that $|x_t|$ then decays like $1/\sqrt{8\eta t}$ rather than geometrically. Why can no fixed learning rate restore a geometric rate on this function?
2. code **Diagonal preconditioning under rotation.** Rotate the valley by an angle θ: $f_\theta(\mathbf{x}) = \frac{1}{2}\mathbf{x}^\top \mathbf{R}_\theta \mathrm{diag}(0.2, 4)\mathbf{R}_\theta^\top \mathbf{x}$, where $\mathbf{R}_\theta$ is the rotation matrix by θ.
    1. For $\theta = 45^\circ$, run plain gradient descent at $\eta = 0.45$, the diagonal preconditioner Equation 9.10 at $\eta = 0.5$, and Newton's method with `d2l.train_2d`. Which of the three behave as on the unrotated valley? Explain the behavior of the diagonal preconditioner from the entries of the rotated Hessian.
    2. The preconditioned update behaves like plain gradient descent on a quadratic with Hessian $\mathbf{D}^{-1/2}\mathbf{H}_\theta\mathbf{D}^{-1/2}$, where $\mathbf{D} = \mathrm{diag}(\mathbf{H}_\theta)$. Compute its condition number as a function of θ, and check that it is 1 at $\theta = 0$ and 20 at $\theta = 45^\circ$.
    3. A deep network's Hessian is not diagonal in its parameter coordinates. What does this exercise suggest about how much a diagonal preconditioner can help, and what would a preconditioner need in order to undo a rotation?
3. code **Absolute-curvature preconditioning.** Replace $\mathrm{diag}(\mathbf{H})^{-1}$ in Equation 9.10 by $|\mathrm{diag}(\mathbf{H})|^{-1}$ and run this variant at $\eta = 1$ on the nonconvex $f(x_1, x_2) = x_1 \cos(cx_1) + x_2 \cos(cx_2)$ with $c = 0.15\pi$ from the starting point of `d2l.train_2d`. Show that every update direction is a descent direction. Does every step decrease f? Explain the size of the largest step of the run.
4. **Certifying the lowest minimum.** ▯ Modify the objective $f(x) = x\cos(cx)$ with $c = 0.15\pi$ of Section 9.2.1.2 so that its local minima in $(-20, 20)$ differ in value by at most 10^{-3} except for one. Explain why any method that uses only f and f' at the points it visits must then visit every basin before it can certify which local minimum is lowest.
5. code **Newton's convergence order.** ▯ In Section 9.2.3, `newton` converges in a few steps on $f(x) = \cosh(cx)$. Run it with $c = 0.5$ from $x_0 = 10$ and record $|x_t|$ after every

step.

   1. From the last steps before x_t reaches 0 in floating point, estimate the order p in $|x_{t+1}| \approx C|x_t|^p$. Where does this fast phase begin, and what governs the steps before it?
   2. Derive p and C from the Taylor expansion of the update $x - \tanh(cx)/c$ around 0, and reconcile p with the quadratic rate stated in Section 35.2.1.

6. code **Stale Hessian factorizations.** ⊠ A Newton step in d dimensions can reuse the Hessian factorization of an earlier step. Minimize $f(\mathbf{x}) = \sum_{i=1}^{n} \cosh(c\,\mathbf{a}_i^\top \mathbf{x})$ with $n = 200$, $d = 50$, $c = 0.5$, directions $\mathbf{a}_i$ with independent standard Gaussian entries, and $\mathbf{x}_0 = \mathbf{1}$. Its minimum is $f^\star = n$ at $\mathbf{x} = \mathbf{0}$. Charge $4nd$ operations per gradient, nd^2 per Hessian evaluation, $d^3/3$ per Cholesky factorization, and $2d^2$ per pair of triangular solves. Evaluate and factor the Hessian on every step, every 5 steps, and every 20 steps, reusing the stale factorization in between. Plot $f(\mathbf{x}_t) - f^\star$ against the cumulative operation count rather than the iteration count. Which variant reaches 10^{-8} first, and which of the four costs dominates its total?

   Adapted from Stephen Boyd and Lieven Vandenberghe, Convex Optimization, Exercise 9.31(a).

9.3 Stochastic and Minibatch Gradient Descent

Every model since Section 2.1 has been trained on minibatches, most of them with minibatch SGD: at each step we drew a few dozen or a few hundred examples, computed the gradient of their average loss, and stepped against it. Section 9.2 studied step size and curvature with the exact gradient, which requires a pass over the whole training set for every step. This section asks what is gained and what is lost when the gradient is estimated from a few examples instead.

The gain is large. A common recipe trains ResNet-50 on the 1.28 million images of ImageNet for 90 epochs in minibatches of 256 images (Goyal et al. 2017), which comes to 450,000 parameter updates. The same number of updates with exact gradients would take 450,000 passes over the data, 5,000 times the work. Yet a few hundred randomly chosen images already indicate which way is downhill. *Stochastic gradient descent* (SGD) uses that observation: it estimates the gradient from a random sample of examples and steps along the estimate. Each step is cheap, and the estimate is right on average, but any single estimate is noisy.

We first ask what the noise does. On a quadratic valley we compute exactly how far from the minimum the iterates of constant-step SGD keep wandering, and how that distance depends on the learning rate, the curvature, and the noise. We then ask what averaging over a minibatch of b examples gains. Averaging divides the variance of the estimate by b, which we measure on a small network. On parallel hardware, moreover, b examples take far less time than b separate steps, and we measure that too.

A race of full-batch, single-example, and minibatch updates against the clock closes the section and builds the training function that the next section uses for momentum. Lowering the learning rate during training is the subject of Section 9.6, and the limits of larger batches that of Section 9.8.

```
import time
```

9.3.1 Stochastic Gradient Descent

9.3.1.1 Sampled Gradients

In deep learning the objective is an average of per-example losses. With n training examples, a parameter vector $\mathbf{x}$, and the loss $f_i(\mathbf{x})$ on example i, the objective and its gradient are

$$f(\mathbf{x}) = \frac{1}{n}\sum_{i=1}^{n} f_i(\mathbf{x}), \qquad \nabla f(\mathbf{x}) = \frac{1}{n}\sum_{i=1}^{n} \nabla f_i(\mathbf{x}). \tag{9.11}$$

One exact gradient therefore costs n per-example gradients. SGD instead draws an index i uniformly at random from $\{1, \dots, n\}$ and steps along that example's gradient,

$$\mathbf{x} \leftarrow \mathbf{x} - \eta \nabla f_i(\mathbf{x}), \tag{9.12}$$

where η is the learning rate. A step now costs one per-example gradient instead of n. The sampled gradient is an *unbiased* estimate of the full gradient: every index has probability $1/n$, so

$$\mathbb{E}_i[\nabla f_i(\mathbf{x})] = \frac{1}{n}\sum_{i=1}^{n} \nabla f_i(\mathbf{x}) = \nabla f(\mathbf{x}). \tag{9.13}$$

On average the stochastic gradient points where the true gradient points. A single draw, however, can point almost anywhere, since one example's gradient pulls toward fitting that example alone. It helps to write the stochastic gradient as the true gradient plus noise, $\nabla f_i(\mathbf{x}) = \nabla f(\mathbf{x}) + \xi$, where the noise ξ has mean zero.

9.3.1.2 SGD on the Valley

To see what the noise does, we return to the valley of Section 9.1,

$$f(\mathbf{x}) = 0.1x_1^2 + 2x_2^2, \tag{9.14}$$

which curves gently along x_1 (second derivative 0.2) and steeply along x_2 (second derivative 4), a condition number of 20. Gradient descent converges on it, slowly along x_1. We simulate sampling noise by adding independent Gaussian noise of unit variance to both coordinates of the exact gradient. Exercise 1 shows that this is SGD on a dataset whose examples are randomly shifted copies of the valley.

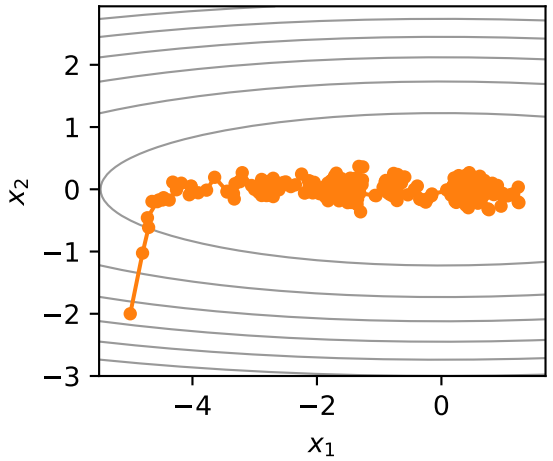
The update rule `sgd_2d` has the form that `d2l.train_2d` of Section 9.2 iterates from the starting point $(-5, -2)$, here with the gradient `d2l.valley_grad` of the valley `d2l.f_valley`. It is the `gd_2d` of that section with noise added to the gradient. The noise comes from a NumPy random generator `rng`, which `train_2d` passes on together with the learning rate.

```
def sgd_2d(x1, x2, s1, s2, f_grad, eta, rng):
    g1, g2 = f_grad(x1, x2)
    n1, n2 = rng.normal(size=2)  # Unit-variance gradient noise
    return x1 - eta * (g1 + n1), x2 - eta * (g2 + n2), 0, 0
```

We run 300 steps at $\eta = 0.1$.

```
rng = np.random.default_rng()
trace = d2l.train_2d(sgd_2d, d2l.valley_grad, steps=300, eta=0.1, rng=rng)
d2l.show_trace_2d(d2l.f_valley, trace)
```

```
step 300, x1: 0.247958, x2: 0.186984
```



For the first hundred or so steps the iterate heads for the minimum much as gradient descent would, slowly along x_1. Then it stops making progress and wanders in a cloud around the minimum, a cloud that is longer along the flat x_1 direction than along the steep x_2 direction. Running longer does not shrink the cloud. Near the minimum the true gradient vanishes but the noise does not, so every step still moves the parameters by η times pure noise.

9.3.1.3 The Noise Ball

How far from the minimum does the cloud extend? On a quadratic the question has an exact answer. Each coordinate of the valley evolves on its own, so take one coordinate with curvature λ (0.2 or 4 here) and gradient noise ξ_t of mean zero and variance σ^2 (1 here), drawn independently at every step. One SGD step reads

$$x_{t+1} = x_t - \eta\,(\lambda x_t + \xi_t) = (1 - \eta\lambda)\,x_t - \eta\,\xi_t. \tag{9.15}$$

The first term pulls the iterate toward the minimum at 0, shrinking it by the factor $1 - \eta\lambda$ as gradient descent would. The second term pushes it in a random direction. To see where the two balance, we square both sides and take expectations. The cross term vanishes because ξ_t has mean zero and is independent of x_t, which leaves

$$\mathbb{E}\big[x_{t+1}^2\big] = (1 - \eta\lambda)^2\,\mathbb{E}\big[x_t^2\big] + \eta^2\sigma^2. \tag{9.16}$$

Far from the minimum the first term dominates, and SGD makes progress like gradient descent. Near it, the second moment settles at the value v at which contraction and injected noise cancel, $v = (1 - \eta\lambda)^2 v + \eta^2\sigma^2$. Solving for v gives

$$\mathbb{E}[x_\infty^2] = \frac{\eta\,\sigma^2}{\lambda\,(2-\eta\lambda)} \approx \frac{\eta\,\sigma^2}{2\lambda} \quad \text{for } \eta\lambda \ll 1. \tag{9.17}$$

The iterates fluctuate in a *noise ball* around the minimum. Its squared radius grows in proportion to the learning rate and to the noise variance, and it shrinks as the curvature grows. The last dependence explains the shape of the cloud. At $\eta = 0.1$ the formula gives a variance of $0.1/(0.2 \cdot 1.98) \approx 0.25$ along x_1 and $0.1/(4 \cdot 1.6) \approx 0.016$ along x_2, so the cloud has standard deviation 0.5 along the flat direction and 0.125 along the steep one, a quarter of that. The flat directions, where gradient descent is slowest, are also where the noise carries the iterate farthest, because the pull back toward the minimum is weakest there.

We check the formula with 10,000 independent SGD runs, the rows of one array, all started at the minimum so that the noise alone decides where they go. After 2,000 steps we measure the variance of each coordinate across the runs, at $\eta = 0.1$ and at half that rate.

```
def noise_ball(eta, lam, num_runs=10000, num_steps=2000):
    """Variance of SGD iterates on sum(lam * x**2 / 2) with unit noise."""
    rng = np.random.default_rng()
    x = np.zeros((num_runs, len(lam)))
    for _ in range(num_steps):
        x = x - eta * (lam * x + rng.normal(size=x.shape))
    return x.var(axis=0)

lam = np.array(d2l.VALLEY_CURVATURES)  # Curvatures of the valley
for eta in (0.1, 0.05):
    print(f'eta={eta}: measured {noise_ball(eta, lam).round(4)}, '
          f'predicted {(eta / (lam * (2 - eta * lam))).round(4)}')
```

```
eta=0.1: measured [0.2542 0.016 ], predicted [0.2525 0.0156]
eta=0.05: measured [0.1264 0.007 ], predicted [0.1256 0.0069]
```

Measured and predicted variances agree to within a few percent, the sampling error of 10,000 runs. Halving the learning rate halves the variance along x_1, as the approximation in Equation 9.17 says. Along x_2 it shrinks by a little more than half, because there $\eta\lambda = 0.4$ is not small and the exact formula applies.

A constant learning rate therefore forces a choice. Halving η halves the squared radius of the ball, but it also halves the fraction $\eta\lambda$ by which each step closes the remaining distance, so reaching the ball takes twice as many steps. Large steps make fast early progress; small steps finish close. The way out is to change η during training, large at first and small at the end. Robbins and Monro (1951) gave the classical conditions under which such a decreasing sequence η_t converges on problems like this one. The rates must sum to infinity, $\sum_t \eta_t = \infty$, so that the iterate can still travel arbitrarily far. Their squares must have a finite sum, $\sum_t \eta_t^2 < \infty$, so that the noise injected over the whole run, whose variances add up to $\sigma^2 \sum_t \eta_t^2$, stays bounded. The rate $\eta_t \propto 1/t$ satisfies both. Section 9.6 develops the schedules used for deep networks, where these conditions are a guide rather than a recipe.

9.3.2 Minibatches

A decaying learning rate shrinks the noise ball by taking smaller steps. Averaging the gradient over several examples shrinks it by reducing the noise itself, and every training loop in

this book does so. We quantify the reduction, and we measure why averaging adds so little time on modern hardware.

9.3.2.1 Gradient Variance and Batch Size

A minibatch $\mathcal{B}_t$ of b examples, drawn independently and uniformly at random, gives the gradient estimate

$$\mathbf{g}_t = \frac{1}{b} \sum_{i \in \mathcal{B}_t} \nabla f_i(\mathbf{x}_t).$$

It is unbiased because every term is. Its noise is smaller. In Section 2.1.2.3 we met this as the proportionality of the noise ball to $1/|\mathcal{B}|$; here we derive the exact constant and use it throughout the chapter. Let Σ be the covariance of a single example's gradient, so that its trace $\operatorname{tr}\Sigma = \mathbb{E}_i\|\nabla f_i(\mathbf{x}) - \nabla f(\mathbf{x})\|^2$ is the expected squared error of a single-example gradient. The errors of independent draws have mean zero and are uncorrelated, so the expected squared norm of their sum is the sum of their expected squared norms, $b \operatorname{tr}\Sigma$. Dividing the sum by b divides this by b^2:

$$\mathbb{E}\left\|\mathbf{g}_t - \nabla f(\mathbf{x}_t)\right\|^2 = \frac{\operatorname{tr}\Sigma}{b}. \tag{9.18}$$

A minibatch of b examples divides the gradient variance by b, and it does so in every coordinate. In the noise ball Equation 9.17 the variance σ^2 becomes σ^2/b, and the squared radius becomes about $\eta\sigma^2/(2\lambda b)$. Halving the learning rate and doubling the batch size shrink the ball by the same factor, at different costs: the first doubles the number of steps needed to reach the ball, the second doubles the work per step.

The derivation holds for any model. To see it on a real one, we need data. The next experiments use a small regression problem, the self-noise of airfoils that NASA measured in a wind tunnel. Each example has five features: the frequency, the angle of attack, the chord length, the free-stream velocity, and the thickness of the boundary layer. The target is the sound pressure level. `get_airfoil` standardizes every column to mean zero and variance one and keeps the first 1,500 examples. It returns a minibatch iterator that reshuffles the examples in each epoch, together with the number of features. A training run on these data takes seconds, so we can repeat it many times.

```
d2l.DATA_HUB['airfoil'] = (d2l.DATA_URL + 'airfoil_self_noise.dat',
                           '76e5be1548fd8222e5074cf0faae75edff8cf93f')


def get_airfoil(batch_size=10, n=1500):
    """Load the standardized airfoil self-noise data as minibatches."""
    data = np.genfromtxt(d2l.download('airfoil'),
                         dtype=np.float32, delimiter='\t')
    data = torch.from_numpy((data - data.mean(axis=0)) / data.std(axis=0))
    dataset = torch.utils.data.TensorDataset(
        *(t.to(d2l.try_gpu()) for t in (data[:n, :-1], data[:n, -1])))
    sampler = torch.utils.data.BatchSampler(  # Whole minibatches of indices
        torch.utils.data.RandomSampler(dataset), batch_size, drop_last=False)
```

```
    data_iter = torch.utils.data.DataLoader(dataset, sampler=sampler,
                                            batch_size=None)
    return data_iter, data.shape[1]-1
```

The data live on the GPU if there is one. A `DataLoader` given a batch size would fetch the examples of a minibatch one at a time and stack them, which on these data takes longer than the update itself. The `BatchSampler` instead draws the indices of a whole minibatch, and the dataset gathers them in a single indexing operation.

On these data we fix a two-layer MLP with 64 hidden units at a random initialization. `batch_grads` computes the gradient of the squared loss, flattened into one vector, for a whole stack of minibatches at once. Given all 1,500 examples as minibatches of one, it returns every per-example gradient, and from those we compute the full gradient and $\operatorname{tr}\Sigma$ exactly.

```
data_iter, num_features = get_airfoil(batch_size=1500)
X, y = next(iter(data_iter))  # The whole dataset
device = X.device
params = {'W1': 0.1 * torch.randn(num_features, 64, device=device),
          'b1': torch.zeros(64, device=device),
          'W2': 0.1 * torch.randn(64, 1, device=device),
          'b2': torch.zeros(1, device=device)}

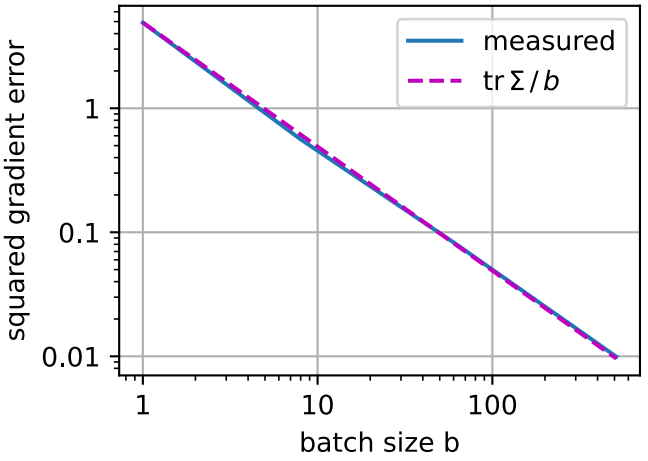
def loss(params, X, y):
    h = torch.relu(X @ params['W1'] + params['b1'])
    y_hat = (h @ params['W2'] + params['b2']).squeeze(-1)
    return (y_hat - y).pow(2).mean() / 2

def flat_grad(params, X, y):
    grads = torch.func.grad(loss)(params, X, y)
    return torch.cat([g.reshape(-1) for g in grads.values()])

batch_grads = torch.func.vmap(flat_grad, in_dims=(None, 0, 0))
G = batch_grads(params, X[:, None], y[:, None])  # (1500, 449)
g_full = G.mean(0)
tr_Sigma = ((G - g_full) ** 2).sum(1).mean()
```

For each batch size from 1 to 512 we now draw 200 minibatches, with replacement as the derivation assumes, and compare their mean squared gradient error with $\operatorname{tr}\Sigma/b$.

```
batch_sizes = [1, 8, 64, 512]
measured = []
for b in batch_sizes:
    idx = torch.randint(0, len(y), (200, b), device=device)
    errors = ((batch_grads(params, X[idx], y[idx]) - g_full) ** 2).sum(1)
    measured.append(errors.mean().item())
d2l.plot(batch_sizes, [measured, [tr_Sigma.item() / b for b in batch_sizes]],
         'batch size b', 'squared gradient error', xscale='log',
         yscale='log', legend=['measured', r'$\mathrm{tr}\,\Sigma\,/\,b$'])
```



The measured errors lie on the predicted line over a factor of 500 in batch size, within the sampling error of 200 draws. The logarithmic axes hide one consequence. Variance falling like $1/b$ means that the size of the noise falls only like $1/\sqrt{b}$: a batch 100 times larger, at 100 times the computation per step, makes the gradient error only 10 times smaller. Whether the larger batch is worth its computation depends on how much the noise still limits the optimizer, which is the question of Section 9.8.

9.3.2.2 Sampling without Replacement

The derivation drew the examples of a minibatch independently, with replacement. Training loops do something else: they shuffle the dataset at the start of every epoch and cut it into consecutive minibatches, so that each example is used exactly once per epoch. Drawing n times with replacement would touch only a fraction $1-(1-1/n)^n \approx 1-e^{-1} \approx 0.63$ of the examples per pass and use others twice or more. Within a minibatch, sampling without replacement only helps: the variance of the average of b distinct examples is

$$\mathbb{E}\left\|\mathbf{g}_t - \nabla f(\mathbf{x}_t)\right\|^2 = \frac{\operatorname{tr}\Sigma}{b} \cdot \frac{n-b}{n-1}, \tag{9.19}$$

where $\operatorname{tr}\Sigma$ averages over the n examples of the dataset. To see where the factor comes from, write $\mathbf{e}_i = \nabla f_i(\mathbf{x}) - \nabla f(\mathbf{x})$ for the error of example i, so that $\operatorname{tr}\Sigma = \frac{1}{n}\sum_i \|\mathbf{e}_i\|^2$. The n errors sum to zero, and so two distinct examples drawn at random have errors that point, on average, slightly against each other:

$$\mathbb{E}[\mathbf{e}_i^\top \mathbf{e}_j] = \frac{1}{n(n-1)} \sum_{i \neq j} \mathbf{e}_i^\top \mathbf{e}_j = \frac{\left\|\sum_i \mathbf{e}_i\right\|^2 - \sum_i \|\mathbf{e}_i\|^2}{n(n-1)} = -\frac{\operatorname{tr}\Sigma}{n-1}. \tag{9.20}$$

The squared norm of the sum of b errors has b terms $\|\mathbf{e}_i\|^2$, each of expectation $\operatorname{tr}\Sigma$, and $b(b-1)$ cross terms, each of expectation $-\operatorname{tr}\Sigma/(n-1)$. They add up to $b\operatorname{tr}\Sigma\,(n-b)/(n-1)$, and dividing by b^2 gives the formula. The correction factor is close to 1 while the batch is a small part of the dataset. It reaches 0 at $b = n$, where the minibatch is the whole dataset and its gradient is exact. Across the minibatches of one epoch, however, the draws

are no longer independent: an example used early in the epoch cannot come back later. The analysis of this section then no longer applies verbatim. Reshuffling usually converges faster in practice, and on strongly convex problems with small learning rates provably so (Mishchenko et al. 2020). Exercise 2 measures the difference.

9.3.2.3 Batched Computation

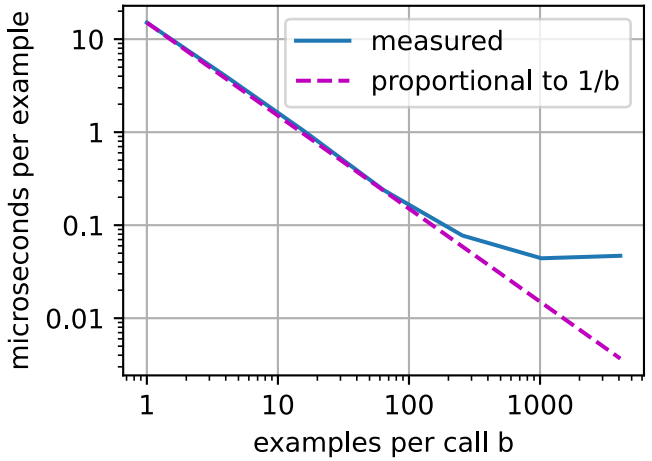
Statistics alone would not make minibatches the default: b examples in one step cost b per-example gradients, as many as b steps of SGD. What makes them the default is the hardware, on which one step over b examples takes far less time than b steps over one. Section 2.1 measured one part of the reason: predicting 1,000 examples in one matrix product is hundreds of times faster than a Python loop over them, because every operation launched from Python carries a fixed overhead of microseconds.

Memory is the other part. A linear layer with a $d \times d$ weight matrix, applied to one example, reads all d^2 weights from memory and performs $2d^2$ floating-point operations, a multiplication and an addition per weight. In single precision, with 4 bytes per weight, that is half an operation per byte read. Applied to a minibatch of b examples, the layer reads the weights once and performs $2bd^2$ operations, about $b/2$ per byte while b is small compared with d. This ratio of operations to bytes moved is the *arithmetic intensity* of a computation. A current GPU performs on the order of a hundred floating-point operations in the time it reads one byte from its memory, a CPU on the order of ten. With one example per call the arithmetic units therefore wait for memory most of the time. Only a batch of tens to hundreds of examples gives them enough work per byte.

We measure the combined effect on such a layer: a 1024×1024 weight matrix applied to 4,096 examples, in blocks of b examples per call, on a GPU if one is available. `d2l.Benchmark` of Section 5.7.2.1 times a function: a warm-up call absorbs one-time costs, and the clock stops only once the device has finished all the timed calls, here five, whose mean time it reports.

```
device = d2l.try_gpu()
W = torch.randn(1024, 1024, device=device)
X = torch.randn(4096, 1024, device=device)
matmul = torch.matmul  # In full float32, PyTorch's default
```

```
batch_sizes = [1, 4, 16, 64, 256, 1024, 4096]
per_example = []
for b in batch_sizes:
    blocks = [X[i:i + b] for i in range(0, len(X), b)]
    timing = d2l.Benchmark(lambda: [matmul(Xb, W) for Xb in blocks],
                           warmup=1, repeats=5)
    per_example.append(timing.time / len(X))
d2l.plot(batch_sizes, [[t * 1e6 for t in per_example],
                       [per_example[0] * 1e6 / b for b in batch_sizes]],
         'examples per call b', 'microseconds per example', xscale='log',
         yscale='log', legend=['measured', 'proportional to 1/b'])
```



The time per example first falls steeply, at small b close to the $1/b$ line. A call takes about the same time whether it carries one example or a few: its time goes to launching it and to reading $\mathbf{W}$, and more examples share that time. Once a call carries enough examples to keep the arithmetic units busy, its time grows with b, and the time per example levels off at the time of the arithmetic itself. Between one example per call and the level part of the curve lies more than an order of magnitude. A minibatch should therefore be at least large enough to reach the level part. The weights of this layer take 4 MB, which fits in the on-chip cache of a current GPU and of many CPUs, so the measurement shows the launch overhead and the bandwidth of the cache more than the speed of main memory. The weights of a large network, gigabytes in all, do not fit; a forward pass streams them from main memory, and the argument above applies directly.

Memory limits the batch from above, since the activations stored for the backward pass grow in proportion to b. When the desired batch does not fit, the gradients of several smaller *micro-batches* can be accumulated before a single update. This reproduces the large batch only if each micro-batch loss is scaled so that the sum is the mean over the whole batch. The update and any gradient clipping must also wait for the last micro-batch. Layers that couple the examples of a batch, batch normalization above all (Section 7.3), still see only the micro-batch. Section 13.4.4 implements this technique.

9.3.3 Minibatch SGD on the Airfoil Data

We now race the extremes and the middle on the airfoil data: full-batch gradient descent, single-example SGD, and minibatches of 10 and 100, each measured against the clock. The race also builds the training function that the next section uses for momentum.

9.3.3.1 The Training Function

The training function takes the optimizer as an update rule with a fixed interface. The rule receives the parameters, a `states` object that holds whatever auxiliary variables the algorithm keeps, and a dictionary of hyperparameters. SGD keeps no state; momentum, in the next section, keeps a running average of past gradients. The training function averages the loss over each minibatch, so the update rule never divides by the batch size.

```
def sgd(params, states, hyperparams):
    for p in params:
        with torch.no_grad():
            p.sub_(hyperparams['lr'] * p.grad)
        p.grad.zero_()
```

The update reads each parameter's gradient from `p.grad`, changes the parameter in place, and clears the gradient for the next minibatch.

To record training time we use a small timer that accumulates intervals. We save it in the `d2l` package, since later chapters use it too.

```
class Timer:
    """Record multiple running times."""
    def __init__(self):
        self.times = []
        self.start()

    def start(self):
        """Start the timer."""
        self.tik = time.perf_counter()

    def stop(self):
        """Stop the timer and record the time in a list."""
        self.times.append(time.perf_counter() - self.tik)
        return self.times[-1]

    def avg(self):
        """Return the average time."""
        return sum(self.times) / len(self.times)

    def sum(self):
        """Return the sum of time."""
        return sum(self.times)

    def cumsum(self):
        """Return the accumulated time."""
        return np.array(self.times).cumsum().tolist()
```

`train_with_rule` trains a linear regression model with a given update rule. Its inner function `step` is the training step proper: the gradient of the loss on a minibatch, then the update rule. The rest is bookkeeping, which `record_run` takes over. It calls `step` on every minibatch, and about every 200 examples it pauses the timer and evaluates the loss on the whole dataset, so the recorded times count training alone. It plots the loss against the epoch unless told not to, and returns a history: the number of examples processed, the cumulative training time, and the loss at each evaluation.

```
def record_run(step, full_loss, data_iter, num_epochs=2, plot=True, *,
               ylim=None):
    """Call step(X, y) on every minibatch; about every 200 examples, record
    full_loss() and the training time so far, and plot the loss in the range
```

```
    ylim. Return the history."""
    # Wait until the device has finished (CUDA runs asynchronously)
    wait = torch.cuda.synchronize if torch.cuda.is_available() else (
        lambda: None)
    if plot:
        animator = d2l.Animator(xlabel='epoch', ylabel='loss',
                                xlim=[0, num_epochs], ylim=ylim)
    history = {'examples': [], 'loss': []}
    n, i, timer = 0, 0, d2l.Timer()
    for _ in range(num_epochs):
        for X, y in data_iter:
            step(X, y)
            n, i = n + len(X), i + 1
            if i % max(1, 200 // len(X)) == 0:  # About every 200 examples
                wait()
                timer.stop()
                with torch.no_grad():
                    l = float(full_loss())
                history['examples'].append(n)
                history['loss'].append(l)
                if plot:
                    animator.add(i / len(data_iter), (l,))
                timer.start()
    history['time'] = timer.cumsum()
    print(f'loss: {l:.3f}, {timer.sum() / num_epochs:.3f} sec/epoch')
    return history
```

```
def train_with_rule(trainer_fn, states, hyperparams, data_iter,
                    feature_dim, num_epochs=2, plot=True,
                    ylim=(0.22, 0.35)):
    """Train linear regression with an update rule; return its history. The
    default ylim is the range of the airfoil losses."""
    Xs, ys = (torch.cat(t) for t in zip(*data_iter))  # The whole data
    w = torch.normal(mean=0.0, std=0.01, size=(feature_dim, 1),
                     device=Xs.device, requires_grad=True)
    b = torch.zeros(1, device=Xs.device, requires_grad=True)
    loss = lambda X, y: d2l.squared_loss(d2l.linreg(X, w, b), y).mean()

    def step(X, y):
        loss(X, y).backward()
        trainer_fn([w, b], states, hyperparams)

    loss(Xs, ys).backward()  # Warm-up, then clear the gradients
    w.grad, b.grad = None, None
    return d2l.record_run(step, lambda: loss(Xs, ys), data_iter, num_epochs,
                          plot, ylim=ylim)
```

The parameters live on the device of the data. Before the timer starts, `train_with_rule` runs one forward and backward pass over the whole dataset, which absorbs one-time costs such as loading the GPU's math libraries. `record_run` waits for the device before it stops the timer, because CUDA returns from a call before the device has finished it.

9.3.3.2 Batch Size and Wall-Clock Time

A smaller batch takes more steps per epoch, and each step is cheaper and noisier than a step on a larger batch. We now race batch sizes against the clock to see which lowers the loss fastest.

`train_sgd` trains with plain SGD at a given learning rate and batch size. We run four configurations. Batch size 1,500 is the whole dataset, which makes it gradient descent with one update per epoch; it runs for 10 epochs. Batch size 1 is single-example SGD with 1,500 updates per epoch. Batch sizes 10 and 100 lie in between. The stochastic runs take 2 epochs each. Their learning rates keep the ratio η/b between 0.004 and 0.005, so by the noise ball their iterates fluctuate about equally. Gradient descent has no noise and can take larger steps, up to the stability limit $\eta < 2/L$ of Section 9.2, where L is the largest curvature of the loss. On these data $L \approx 2.1$, so the limit is about 0.95, and we run gradient descent at 0.9.

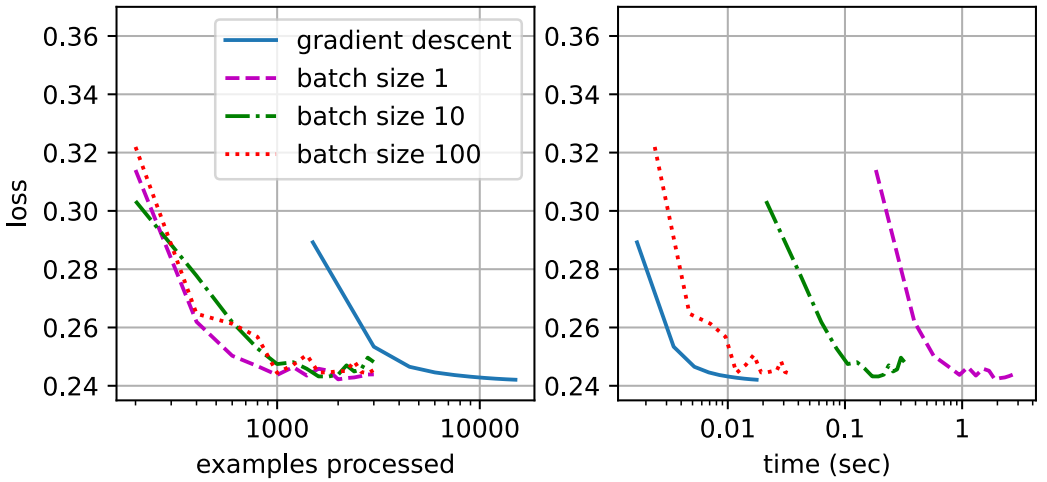
```
def train_sgd(lr, batch_size, num_epochs=2):
    data_iter, feature_dim = get_airfoil(batch_size)
    return train_with_rule(sgd, None, {'lr': lr}, data_iter, feature_dim,
                           num_epochs, plot=False)

runs = {'gradient descent': train_sgd(0.9, 1500, 10),
        'batch size 1': train_sgd(0.005, 1),
        'batch size 10': train_sgd(0.05, 10),
        'batch size 100': train_sgd(0.4, 100)}
```

```
loss: 0.242, 0.002 sec/epoch
loss: 0.244, 1.468 sec/epoch
loss: 0.248, 0.161 sec/epoch
loss: 0.245, 0.018 sec/epoch
```

All four runs end at about the same loss, so the race is decided by time. The printed times per epoch already show the hardware effect. Every run processes the same 1,500 examples per epoch, but single-example SGD does so in 1,500 steps, against 150 and 15 for the batches of 10 and 100. Each step launches its computations separately, however little work they hold, so single-example SGD takes by far the longest per epoch. We plot the loss twice: against the number of examples processed, which measures how much each method extracts from the data, and against the training time, which also reflects how well each method uses the hardware.

```
fig, axes = d2l.plt.subplots(1, 2, figsize=(6.5, 2.8))
for ax, x, xlabel, ylabel, legend in zip(
        axes, ['examples', 'time'], ['examples processed', 'time (sec)'],
        ['loss', None], [list(runs), None]):
    d2l.plot([h[x] for h in runs.values()], [h['loss'] for h in runs.values()],
             xlabel, ylabel, legend, ylim=[0.235, 0.37], xscale='log',
             axes=ax)
```



Measured in examples, the three stochastic runs progress at about the same rate, since they take about the same step η/b per example. Gradient descent lags far behind: after one pass over the data it has made one update, the others 15 to 1,500. Measured in seconds, single-example SGD is the slowest, because it computes its gradients one example at a time, and the batch of 10 lies between it and the batch of 100. Only the batch of 100 is among the fastest in both views. Gradient descent even pulls ahead of it on the clock, but only because the dataset is tiny, so that a full gradient takes no longer than a few minibatch steps. With millions of examples, one full gradient would take as long as thousands of minibatch steps, and gradient descent would lag on the clock as well.

9.3.3.3 Concise Implementation

Every framework ships these update rules as optimizer objects. `train_with_optimizer` is the counterpart of `train_with_rule` for them. It trains the same linear model with the same initialization and the same squared loss, including the factor $\frac{1}{2}$ of Section 2.1. The built-in mean squared error of most libraries omits that factor, which would double the gradient and change the meaning of a learning rate.

```
def train_with_optimizer(trainer_fn, hyperparams, data_iter, num_epochs=2,
                         plot=True, ylim=(0.22, 0.35)):
    """Train linear regression with a torch.optim optimizer class; return
    its history. The default ylim is the range of the airfoil losses."""
    Xs, ys = (torch.cat(t) for t in zip(*data_iter))  # The whole data
    net = nn.Linear(Xs.shape[1], 1).to(Xs.device)
    nn.init.normal_(net.weight, std=0.01)
    nn.init.zeros_(net.bias)
    optimizer = trainer_fn(net.parameters(), **hyperparams)
    loss = lambda X, y: d2l.squared_loss(net(X), y).mean()

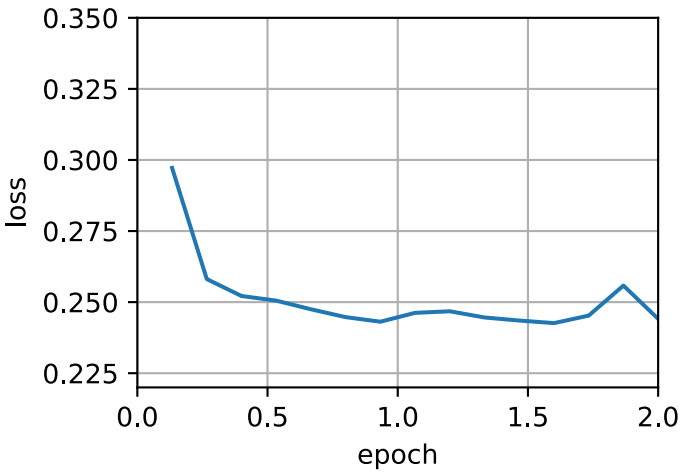
    def step(X, y):
        optimizer.zero_grad()
        loss(X, y).backward()
        optimizer.step()
```

```
    loss(Xs, ys).backward()  # Warm-up; zero_grad clears it
    return d2l.record_run(step, lambda: loss(Xs, ys), data_iter, num_epochs,
                          plot, ylim=ylim)
```

Repeating the run with batch size 10 and learning rate 0.05 through the framework's optimizer reproduces the from-scratch result.

```
data_iter, _ = get_airfoil(10)
train_with_optimizer(torch.optim.SGD, {'lr': 0.05}, data_iter);
```

```
loss: 0.244, 0.129 sec/epoch
```



We can now predict where constant-step SGD stalls and how a minibatch moves that point, and we have measured why minibatches add so little time on parallel hardware. Three questions remain. How should the learning rate fall during training? How large can the batch grow before a larger one stops reducing the number of steps? And how can SGD move faster along the flat directions of an ill-conditioned valley, where gradient descent is slow and the noise ball is widest? Later sections of this chapter take up the first two questions. The next section answers the third with momentum.

9.3.4 Summary

Stochastic gradient descent replaces the exact gradient, which requires a pass over all n examples, with an unbiased estimate from a random sample. With a constant learning rate η the iterates do not converge. They fluctuate in a noise ball whose squared radius is about $\eta\sigma^2/(2\lambda)$ in a direction of curvature λ, so the ball is widest where the objective is flattest. Decreasing learning rates converge if their sum diverges and the sum of their squares does not. A minibatch of b examples divides the gradient variance by b, which we measured over a factor of 500 in batch size. On parallel hardware it also takes far less time than b single-example steps, because the computation becomes limited by arithmetic rather than by memory and launch overhead. Of gradient descent, single-example SGD, and minibatch SGD, only the last is fast both per example processed and per second.

9.3.5 Exercises

1. **Noise as a random dataset.** The simulation in `sgd_2d` adds Gaussian noise to the gradient of $f(\mathbf{x}) = 0.1x_1^2 + 2x_2^2$.
   1. Show that this is SGD on the per-example losses $f(\mathbf{x}, \mathbf{w}) = 0.1(x_1 - w_1)^2 + 2(x_2 - w_2)^2$, with $\mathbf{w}$ drawn from a normal distribution with mean zero, and find the covariance of $\mathbf{w}$ that reproduces unit-variance noise in both gradient coordinates.
   2. Why must the examples be spread much more widely along x_1 than along x_2? How does the expected loss $\mathbb{E}_{\mathbf{w}} f(\mathbf{x}, \mathbf{w})$ differ from the valley?
2. code **Reshuffling versus replacement.** ⊠ Draw a dataset of $n = 100$ vectors $\mathbf{w}_i$ from the distribution of the previous problem and run SGD on the resulting finite sum for 20 epochs from the starting point $(-5, -2)$, once sampling i with replacement and once reshuffling the dataset before each epoch. Record the distance to the minimizer of the finite sum after each epoch, averaged over 20 runs, at the constant learning rates $\eta = 0.1$ and $\eta = 0.01$. At which learning rate does reshuffling make the larger difference, and why?
3. code **Per-coordinate learning rates.** ⊠ The noise ball Equation 9.17 applies to each coordinate of the valley separately, with curvatures $\lambda_1 = 0.2$ and $\lambda_2 = 4$.
   1. Give each coordinate its own learning rate $\eta_j = \eta/\lambda_j$. Derive the contraction factor per step and the stationary variance of each coordinate, and compare the number of steps gradient descent needs to shrink both coordinates by a factor of 100 with that of a common rate.
   2. Implement the per-coordinate rates in `sgd_2d` and in `noise_ball` with $\eta = 0.1$, and verify both predictions numerically.
   3. Why does this choice need information that a single stochastic gradient does not supply?
4. code **Gradient variance after training.** ⊠ The measurement in Section 9.3.2.1 froze the two-layer network at a random initialization. Train it on the airfoil data for ten epochs of minibatch SGD with batch size 10 and learning rate 0.05, reshuffling the examples before each epoch. At initialization and again after training, compute `g_full` and $\operatorname{tr}\Sigma$ from the per-example gradients. Split $\operatorname{tr}\Sigma$ into the contributions of `W1`, `b1`, `W2`, and `b2`, and explain the changes in the `b2` and `W1` contributions from the form of the per-example gradient of the squared loss.
5. **Duplicated data.** Suppose every example in the training set is silently duplicated, so that the dataset doubles in size while its empirical distribution is unchanged. For SGD with $b = 1$, for minibatch SGD at a fixed b, and for full-batch gradient descent, state whether the cost per step, the number of steps to reach a given loss, and the variance of the gradient estimate change, and explain each answer.
6. code **Where the time per example levels off.** ⊠ The measurement in Section 9.3.2.3 applied a 1024×1024 weight matrix to blocks of b examples.
   1. Repeat it for $d \times d$ weight matrices with $d \in \{256, 1024, 4096\}$. For each d, find the smallest b at which the time per example is within 20% of its minimum.
   2. Including the bytes of the inputs and outputs, the arithmetic intensity of the layer in single precision is $b/(2 + 4b/d)$. Derive this expression, and use it to predict how the b you found should depend on d. Does the measurement agree?
   3. On a GPU, repeat the measurement for $d = 1024$ in bfloat16. Which way does the bend move, and which two properties of the format explain the move?
7. **Memory traffic of matrix multiplication.** Computing $\mathbf{A} = \mathbf{BC}$ for $n \times n$ matrices takes $2n^3$ floating-point operations, however they are ordered. Count the floats each

of three orderings reads from main memory, assuming a cache that holds M floats with $n < M \ll n^2$, and keep only the terms of order n^3.

1. Computing every entry as a dot product of a row of **B** with a column of **C** reads both for every entry. Computing **A** one column at a time keeps the column $\mathbf{C}_{:,j}$ in cache while it reads all of **B**. Show that the second ordering halves the traffic of the first, and compute the arithmetic intensity of both in single precision.
2. A blocked ordering splits the matrices into $w \times w$ blocks. It keeps one block of **A** in cache while it accumulates the n/w products of a block of **B** with a block of **C** that contribute to it. Show that it reads about $2n^3/w$ floats, and find the largest w for which the three blocks fit in the cache.
3. How many bytes of cache does the blocked ordering need to reach an arithmetic intensity of 100 operations per byte in single precision?

8. **Unequal micro-batches.** A training loop accumulates the gradients of $k = 4$ micro-batches into one update with an effective batch of 100 examples, drawn uniformly at random from the training set. The micro-batches hold 32, 32, 32, and 4 examples. The loop averages the loss of each micro-batch over its own examples and divides the accumulated gradient by k. Assume that the per-example gradients are independent, with mean $\nabla f(\mathbf{x})$ and covariance Σ.
   1. Find the weight that the accumulated gradient gives to each example, and show that it is still an unbiased estimate of $\nabla f(\mathbf{x})$.
   2. Express the covariance of the accumulated gradient as a multiple of Σ, and find the batch size of a uniformly weighted average with the same covariance.
   3. State the scaling of the micro-batch losses that makes the accumulated gradient the average over all 100 examples. Use Equation 9.17 to compare the squared radius of the noise ball of the two loops at the same learning rate.
9. code **Batch size and learning rate together.** ☒ `train_sgd` trains the linear model at a chosen batch size and learning rate and returns its history: the examples processed, the cumulative training time, and the loss at each evaluation. Fix $\eta = 0.05$ at $b = 10$ and derive the learning rate for each $b \in \{1, 10, 100, 1000\}$ by two rules, $\eta \propto \sqrt{b}$ and $\eta \propto b$.
   1. Train each configuration for two epochs. Report the time per epoch and the final loss of each run, and name the runs that diverge.
   2. The training loss is a quadratic in the weights and the bias. Its Hessian is

$$\mathbf{H} = \frac{1}{n} \sum_{i=1}^{n} \tilde{\mathbf{x}}_i \tilde{\mathbf{x}}_i^\top, \tag{9.21}$$

   where $\tilde{\mathbf{x}}_i$ appends a constant 1 to the features of example i and $n = 1500$. Compute the largest eigenvalue L of **H** from the standardized data that `get_airfoil` loads, and confirm the value quoted for the race. Gradient descent on a quadratic is stable only for $\eta < 2/L$ (Section 9.2); use this to explain which runs diverge.
   3. At $b = 1$, which rule reaches the lower final loss? Explain the ordering with Equation 9.17.

9.4 Momentum

Several of the convolutional networks in this book were trained by SGD with `momentum=0.9` (Section 7.6). Adam, in the next section, has momentum built in, and so does almost every other optimizer in deep learning. We have used the setting without asking what it does.

Momentum addresses the weakness of gradient descent that Section 9.1 and Section 9.2 measured on a narrow valley: a learning rate small enough for the steep direction makes little progress along the flat one. This section asks how much faster an optimizer can go if it remembers its past gradients.

The valley $f(\mathbf{x}) = 0.1x_1^2 + 2x_2^2$ shows how much there is to gain. At its best learning rate, gradient descent needs 47 steps to shrink both coordinates to 1% of their starting values. The steep direction, with curvature 4, keeps the learning rate below $2/4 = 0.5$. At such a rate the flat direction, with curvature 0.2, shrinks by less than a tenth per step. The number of steps grows in proportion to the condition number, here $\kappa = 20$, and the condition numbers of deep networks run into the hundreds and beyond.

Gradient descent is this slow because it looks only at the current gradient. The gradients of the last few steps say more. Along the floor of the valley they point the same way, step after step; across the valley they alternate in sign, because each step overshoots the opposite wall. *Momentum* keeps a running average of past gradients and steps along the average instead. The average grows along the floor and cancels across the walls. With well-chosen settings it needs 40 steps on the valley where gradient descent needs 139 to reach a millionth of the starting distance, and in general a number of steps proportional to $\sqrt{\kappa}$ rather than κ. Momentum needs one extra buffer per parameter and one more hyperparameter.

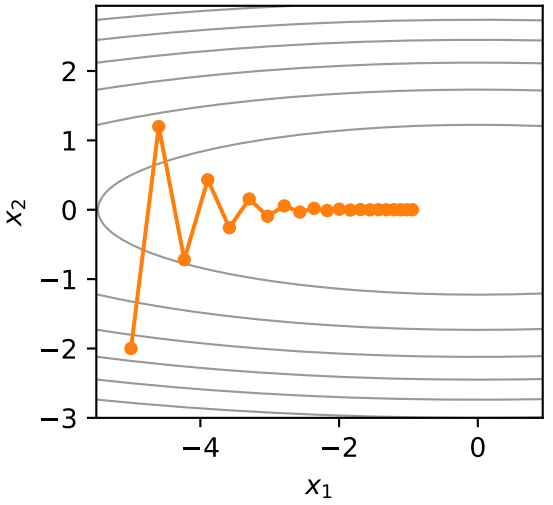
We derive the method on the valley, work out how to set its hyperparameter, and implement it for the minibatch training of Section 9.3.

9.4.1 An Ill-Conditioned Valley

The valley `d2l.f_valley` of Section 9.2 curves gently along x_1, with second derivative $h_1 = 0.2$, and steeply along x_2, with $h_2 = 4$. Its gradient `d2l.valley_grad` is $(h_1 x_1, h_2 x_2)$. As in Section 9.2, an update rule for `d2l.train_2d` receives the two coordinates, two state variables, the gradient function, and the hyperparameters, and returns the new coordinates and state; `train_2d` runs it from $(-5, -2)$ and `d2l.show_trace_2d` draws the path over the contour lines. Gradient descent, `d2l.gd_2d`, ignores the state. We run it at $\eta = 0.4$:

```
h1, h2 = d2l.VALLEY_CURVATURES  # Curvatures of the valley along x1 and x2
f_valley, valley_grad = d2l.f_valley, d2l.valley_grad
d2l.show_trace_2d(f_valley, d2l.train_2d(d2l.gd_2d, valley_grad, eta=0.4))
```

```
step 20, x1: -0.943467, x2: -0.000073
```



In each coordinate a step of gradient descent multiplies the distance to the minimum by $1 - \eta h$. At $\eta = 0.4$ the factor is 0.92 along x_1, so after 20 steps x_1 has only shrunk from -5 to about -0.94. Along x_2 the factor is $1 - 1.6 = -0.6$: the iterate overshoots the floor and lands on the opposite wall, closer to the minimum each time. A larger learning rate would speed up x_1, but it would push the factor of x_2 below -1. At $\eta = 0.6$ it is $1 - 2.4 = -1.4$, and every step throws x_2 farther up the opposite wall than the last:

```
d2l.train_2d(d2l.gd_2d, valley_grad, eta=0.6);
```

```
step 20, x1: -0.387814, x2: -1673.365109
```

After 20 steps x_2 has grown from -2 to about -1673, while x_1 reached -0.39, more than twice as close as at $\eta = 0.4$. The history of gradients separates the two directions. Along x_1 successive gradients agree: they are small, but they all point the same way. Along x_2 they alternate in sign, and each step undoes most of the last. An average over past gradients would amplify the first and cancel the second.

9.4.2 The Momentum Method

9.4.2.1 Leaky Averages

A minibatch averages gradients over *examples* (Section 9.3). Momentum also averages them over *time*, with an average that leaks: each past gradient is discounted by a factor β per step. The update replaces the gradient by a *velocity* $\mathbf{v}_t$,

$$\begin{aligned} \mathbf{v}_t &\leftarrow \beta \mathbf{v}_{t-1} + \mathbf{g}_t, \\ \mathbf{x}_t &\leftarrow \mathbf{x}_{t-1} - \eta \mathbf{v}_t, \end{aligned} \tag{9.22}$$

where $\beta \in [0, 1)$, the velocity starts at $\mathbf{v}_0 = \mathbf{0}$, and $\mathbf{g}_t$ is the gradient at $\mathbf{x}_{t-1}$: exact, from one example, or from a minibatch. For $\beta = 0$ the update is gradient descent. Unrolling the first line shows what the velocity holds,

$$\mathbf{v}_t = \sum_{\tau=0}^{t-1} \beta^\tau \mathbf{g}_{t-\tau}.$$

The velocity is an exponentially weighted sum of all past gradients, in which the gradient of τ steps ago counts with weight β^τ. The name comes from mechanics. A heavy ball rolling down the objective accumulates the forces acting on it instead of following the slope of the moment, and $1 - \beta$ plays the role of friction. The method is *heavy-ball momentum* (Polyak 1964). Sutskever et al. (2013) showed how much it matters for training deep networks.

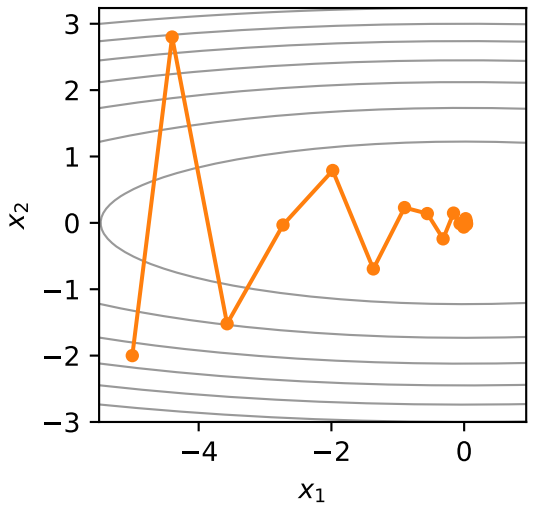
9.4.2.2 Momentum on the Valley

The update rule `momentum_2d` keeps the velocity in the two state variables of `train_2d`. At learning rate 0.6, where gradient descent diverged, we add momentum $\beta = 0.5$:

```
def momentum_2d(x1, x2, v1, v2, f_grad, eta, beta):
    g1, g2 = f_grad(x1, x2)
    v1, v2 = beta * v1 + g1, beta * v2 + g2
    return x1 - eta * v1, x2 - eta * v2, v1, v2

d2l.show_trace_2d(f_valley, d2l.train_2d(momentum_2d, valley_grad, eta=0.6,
                                         beta=0.5))
```

```
step 20, x1: 0.007188, x2: 0.002553
```



The first steps still swing across the valley, but the swings die out within about ten steps, and the iterate then runs down the floor. After 20 steps both coordinates are within 0.01 of the minimum. The velocity did what the history of gradients suggested: the alternating components along x_2 partly cancel, which tames the overshoot, while the persistent components along x_1 add up.

Nothing in Equation 9.22 requires the gradient to be exact. With minibatch gradients the same average also smooths the sampling noise over steps, without any extra gradient eval-

uations, so the velocity is less noisy than a single minibatch gradient. Whether the iterates become less noisy too depends on how η changes along with β, the question of Exercise 6.

9.4.2.3 The Timescale of β

How much history does the velocity hold? The weights $1, \beta, \beta^2, \ldots$ sum to $1/(1-\beta)$, so momentum β averages over roughly the last $1/(1-\beta)$ gradients. We compute this window for three common values, together with the half-life $\log \frac{1}{2} / \log \beta$, the number of steps after which a gradient's weight has halved, and the share of the total weight that falls on the gradients inside the window.

```
print('beta   window   half-life   weight inside the window')
for beta in [0.5, 0.9, 0.99]:
    window = 1 / (1 - beta)
    half_life = np.log(0.5) / np.log(beta)
    inside = 1 - beta ** window
    print(f'{beta:4}   {window:6.0f}   {half_life:9.1f}   {inside:24.2f}')
```

```
beta   window   half-life   weight inside the window
 0.5        2         1.0                      0.75
 0.9       10         6.6                      0.65
0.99      100        69.0                      0.63
```

The value $\beta = 0.9$, the `momentum=0.9` that we have passed to optimizers since Section 7.6, looks back about 10 steps, and $\beta = 0.99$ about 100. The gradients inside the window carry between 63% and 75% of the total weight. As β approaches 1 the share approaches $1-e^{-1} \approx 0.63$, because $\beta^{1/(1-\beta)} \to e^{-1}$.

The same sum fixes the length of a step. When successive gradients agree, the velocity builds up to $1/(1-\beta)$ times a typical gradient, so in persistent directions momentum takes steps of effective size

$$\eta_{\text{eff}} = \frac{\eta}{1-\beta}. \tag{9.23}$$

Raising β from 0.9 to 0.99 at a fixed η therefore makes the steps ten times longer, not only smoother. When β goes up, η has to come down, a rule we test on real data below.

9.4.2.4 Acceleration and Damping

On an ill-conditioned quadratic, momentum is not only more stable than gradient descent but provably faster. Let the Hessian have eigenvalues between $\lambda_{\min}$ and $\lambda_{\max}$, with condition number $\kappa = \lambda_{\max}/\lambda_{\min}$. At its best learning rate, gradient descent shrinks the error by the factor $(\kappa-1)/(\kappa+1)$ per step (Section 9.1), and so needs about $\frac{\kappa}{2} \log \frac{1}{\epsilon}$ steps to reduce it by a factor ϵ. Heavy-ball momentum with the settings

$$\eta^\star = \frac{4}{\left(\sqrt{\lambda_{\max}} + \sqrt{\lambda_{\min}}\right)^2}, \qquad \beta^\star = \left(\frac{\sqrt{\kappa}-1}{\sqrt{\kappa}+1}\right)^2 \tag{9.24}$$

shrinks it by the factor $\sqrt{\beta^\star} = (\sqrt{\kappa}-1)/(\sqrt{\kappa}+1)$ per step, and so needs about $\frac{\sqrt{\kappa}}{2}\log\frac{1}{\epsilon}$ steps (Polyak 1964). The square root of the condition number takes the place of the condition number. For $\kappa = 100$ that saves a factor of 10 with $\beta^\star \approx 0.67$; for $\kappa = 10^4$ a factor of 100 with $\beta^\star \approx 0.96$. The worse the conditioning, the closer the best β moves to 1. Since $\beta^\star \approx 1-4/\sqrt{\kappa}$ for large κ, the timescale reading of Section 9.4.2.3 says that the best average spans about $\sqrt{\kappa}/4$ steps: 25 steps for $\kappa = 10^4$.

On the valley, $\kappa = 20$ gives $\eta^\star \approx 0.67$ and $\beta^\star \approx 0.40$. We check the prediction by counting steps. For each setting the code below runs `momentum_2d` and counts the steps after which both coordinates stay within 1%, and within 10^{-6}, of their starting distance from the minimum. It also computes the contraction factor, the largest factor by which a mode of the valley shrinks per step. In a mode of curvature λ the coordinate obeys $x_{t+1} = (1+\beta-\eta\lambda)\,x_t - \beta x_{t-1}$, a recurrence whose solutions are powers z^t of the roots z of $z^2-(1+\beta-\eta\lambda)z+\beta = 0$; the factor is the largest root in absolute value over the two curvatures.

```
def steps_to(tol, eta, beta, steps=400):
    """Steps after which |x1|, |x2| stay below tol times their start."""
    x1, x2, v1, v2, last = -5.0, -2.0, 0.0, 0.0, 0
    for t in range(1, steps + 1):
        x1, x2, v1, v2 = momentum_2d(x1, x2, v1, v2, valley_grad, eta, beta)
        if abs(x1) > 5 * tol or abs(x2) > 2 * tol:
            last = t
    return last + 1

def contraction(eta, beta):
    return max(abs(np.roots([1, -(1 + beta - eta * lam), beta])).max()
               for lam in (h1, h2))

eta_opt = 4 / (np.sqrt(h2) + np.sqrt(h1)) ** 2
beta_opt = ((np.sqrt(h2 / h1) - 1) / (np.sqrt(h2 / h1) + 1)) ** 2
print('  eta   beta   factor   steps to 1%   steps to 1e-6')
for eta, beta in [(2 / (h1 + h2), 0), (0.6, 0.5), (eta_opt, beta_opt),
                  (0.3, 0.8)]:
    print(f'{eta:5.3f}  {beta:5.3f}  {contraction(eta, beta):7.3f}  '
          f'{steps_to(1e-2, eta, beta):12d}  {steps_to(1e-6, eta, beta):14d}')
```

```
  eta   beta   factor   steps to 1%   steps to 1e-6
0.476  0.000    0.905            47             139
0.600  0.500    0.707            16              42
0.668  0.403    0.635            18              40
0.300  0.800    0.894            42             126
```

The first row is gradient descent at its best rate $2/(h_1+h_2)$, with factor 0.905. Tuned momentum shrinks the error by 0.635 per step and needs 40 steps to reach 10^{-6} where gradient descent needs 139. Per step, the logarithm of its factor is $\log 0.635/\log 0.905 \approx 4.5$ times larger, which is $\sqrt{\kappa}$; the step counts differ by a smaller ratio, 3.5, because the factor describes the error only after the first few steps. The setting $(0.6, 0.5)$ of the plot above is nearly as fast, and to reach 1% it is even faster.

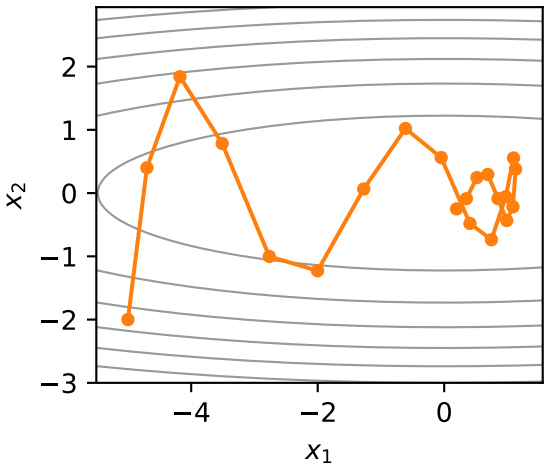
The last row raises β to 0.8 and halves the learning rate to a value at which gradient descent alone would be stable. Its factor 0.894 is barely better than that of gradient descent.

A model of the iteration explains why. In each eigendirection of the Hessian, the recurrence

above is a mass on a spring with friction $1 - \beta$. With too little momentum the system is *overdamped*: both roots are real and one of them is close to 1, so the iterate creeps down the valley as gradient descent does. With too much momentum it is *underdamped*: the roots are complex, both have size $\sqrt{\beta}$, and the iterate overshoots and rings around the minimum. The fastest setting sits between the two, at *critical damping*, where the two roots coincide. For a mode of curvature λ at a fixed η this happens at the critical value $\beta_c = (1 - \sqrt{\eta\lambda})^2$. At $\eta^\star$ the critical values of the flattest and of the steepest mode both equal $\beta^\star$, which is why Equation 9.24 is the best setting for all modes at once. At $\eta = 0.3$ the two critical values of the valley are 0.57 and 0.01, so $\beta = 0.8$ is underdamped in both. We plot the run.

```
d2l.show_trace_2d(f_valley, d2l.train_2d(momentum_2d, valley_grad, eta=0.3,
                                         beta=0.8))
```

```
step 20, x1: 0.195696, x2: -0.249649
```



The iterate swings across the valley for many steps, and along the floor it overshoots the minimum and swings back before it settles. This is a different failure from the divergence of gradient descent at $\eta = 0.6$: the iterates stay bounded and converge, only slowly. Figure 9.6 draws the trade-off for a single mode. The contraction factor falls as β grows, is smallest at the critical value β_c, and past it rises again as $\sqrt{\beta}$.

The $\sqrt{\kappa}$ law is a statement about quadratics. On general objectives it holds near a minimum, where the quadratic approximation of the loss applies, and the evidence that momentum speeds up the training of deep networks is empirical (Sutskever et al. 2013). Section 35.1.3 proves the law and states the matching lower bound: no method built from gradients has a better dependence on κ in the worst case.

9.4.3 Implementation

9.4.3.1 From Scratch

Compared with minibatch SGD, momentum keeps one buffer per parameter, the velocity, of the same shape as the parameter. The training function `d2l.train_with_rule` of Section 9.3.3 provides the `states` argument for it. That function trains a linear regression model on the airfoil data with a given update rule and returns the loss over the examples processed.

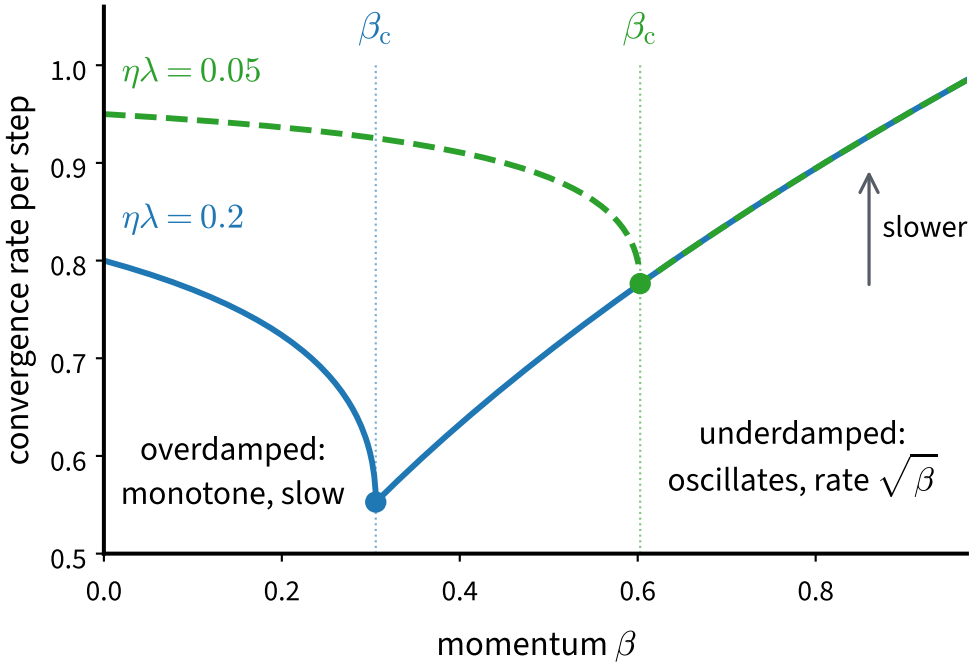


Figure 9.6: Convergence rate per step, the contraction factor of the error, of heavy-ball momentum on a single quadratic mode, as a function of the momentum β. Below the critical value $\beta_c = (1 - \sqrt{\eta\lambda})^2$ the iteration is overdamped and slow; at β_c it is fastest; beyond it the rate degrades gently as $\sqrt{\beta}$. Worse conditioning (smaller $\eta\lambda$, dashed) pushes β_c toward one, which is why large momentum values are the common default.

```
def init_momentum_states(feature_dim):
    device = d2l.try_gpu()  # Where get_airfoil puts the data
    return (d2l.zeros((feature_dim, 1), device=device),
            d2l.zeros(1, device=device))
```

```
def sgd_momentum(params, states, hyperparams):
    for p, v in zip(params, states):
        with torch.no_grad():
            v[:] = hyperparams['momentum'] * v + p.grad
            p[:] -= hyperparams['lr'] * v
        p.grad.zero_()
```

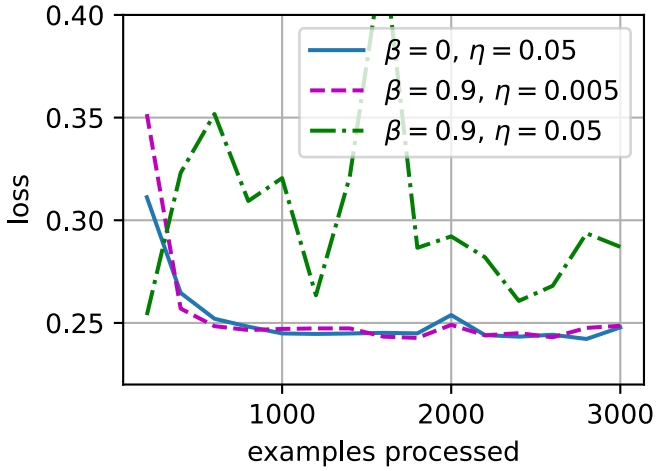
The update changes each velocity and each parameter in place and clears the gradient for the next minibatch, as sgd of Section 9.3 does.

The effective step $\eta/(1-\beta)$ predicts how momentum interacts with the learning rate. We train with minibatches of 10 examples for two epochs in three configurations: plain SGD at $\eta = 0.05$; momentum $\beta = 0.9$ at $\eta = 0.005$, the same effective step; and $\beta = 0.9$ at $\eta = 0.05$, ten times the effective step.

```
def train_momentum(lr, momentum, num_epochs=2):
    data_iter, feature_dim = d2l.get_airfoil(batch_size=10)
    return d2l.train_with_rule(sgd_momentum, init_momentum_states(feature_dim),
                               {'lr': lr, 'momentum': momentum}, data_iter,
                               feature_dim, num_epochs, plot=False)

runs = {r'$\beta = 0$, $\eta = 0.05$': train_momentum(0.05, 0),
        r'$\beta = 0.9$, $\eta = 0.005$': train_momentum(0.005, 0.9),
        r'$\beta = 0.9$, $\eta = 0.05$': train_momentum(0.05, 0.9)}
d2l.plot([h['examples'] for h in runs.values()],
         [h['loss'] for h in runs.values()], 'examples processed', 'loss',
         legend=list(runs), ylim=[0.22, 0.4])
```

```
loss: 0.248, 0.545 sec/epoch
loss: 0.249, 0.542 sec/epoch
loss: 0.287, 0.541 sec/epoch
```



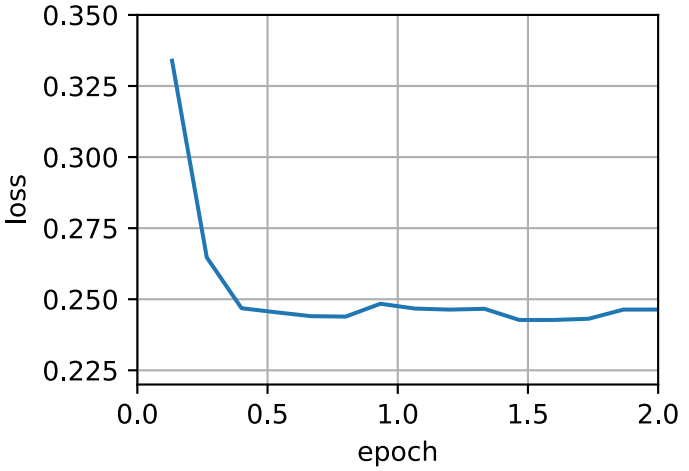
The two runs with the same effective step are almost indistinguishable. On this small, noisy problem momentum brings no speedup: the loss is limited by the noise of the minibatch gradients, and momentum at a matched effective step leaves the iterates about as noisy as plain SGD (Exercise 6 shows why). The third run keeps η and adds momentum, which multiplies the effective step by ten, and its loss stays higher and fluctuates more. The rule of Section 9.4.2.3 holds: when β goes up, η comes down by the factor $1 - \beta$.

9.4.3.2 Concise Implementation

Momentum is a single argument of the SGD optimizer in every framework. `d2l.train_with_optimizer` of Section 9.3 trains the same linear model with a framework optimizer. At $\eta = 0.005$ and $\beta = 0.9$ it reproduces the from-scratch run.

```
data_iter, _ = d2l.get_airfoil(batch_size=10)
d2l.train_with_optimizer(torch.optim.SGD, {'lr': 0.005, 'momentum': 0.9},
                         data_iter);
```

```
loss: 0.246, 0.537 sec/epoch
```



`torch.optim.SGD` implements Equation 9.22 as written: with its default `dampening=0` it adds each gradient to β times the stored velocity.

9.4.4 Nesterov Momentum

The ringing at $\beta = 0.8$ has a cause we can name. The velocity carries the iterate past the minimum, and the gradient computed at the current point only notices the overshoot one step after it has happened. Nesterov (1983) proposed to evaluate the gradient at the point the velocity is about to carry the iterate to:

$$\begin{aligned} \mathbf{v}_t &\leftarrow \beta \mathbf{v}_{t-1} + \nabla f(\mathbf{x}_{t-1} - \eta\beta\,\mathbf{v}_{t-1}), \\ \mathbf{x}_t &\leftarrow \mathbf{x}_{t-1} - \eta\,\mathbf{v}_t. \end{aligned} \tag{9.25}$$

The point $\mathbf{x}_{t-1} - \eta\beta\,\mathbf{v}_{t-1}$ is where the old velocity alone would take the iterate. If that point lies past the floor of the valley, the gradient there already points back, and the velocity is corrected before the overshoot rather than after it. Figure 9.7 shows one step of each method from the same point and velocity.

In code the change is the point at which the gradient is taken:

```
def nesterov_2d(x1, x2, v1, v2, f_grad, eta, beta):
    g1, g2 = f_grad(x1 - eta * beta * v1,  # Gradient at the look-ahead
                    x2 - eta * beta * v2)  # point, not at (x1, x2)
    v1, v2 = beta * v1 + g1, beta * v2 + g2
    return x1 - eta * v1, x2 - eta * v2, v1, v2

d2l.show_trace_2d(f_valley, d2l.train_2d(nesterov_2d, valley_grad, eta=0.3,
                                         beta=0.8))
```

```
step 20, x1: 0.217791, x2: -0.000070
```

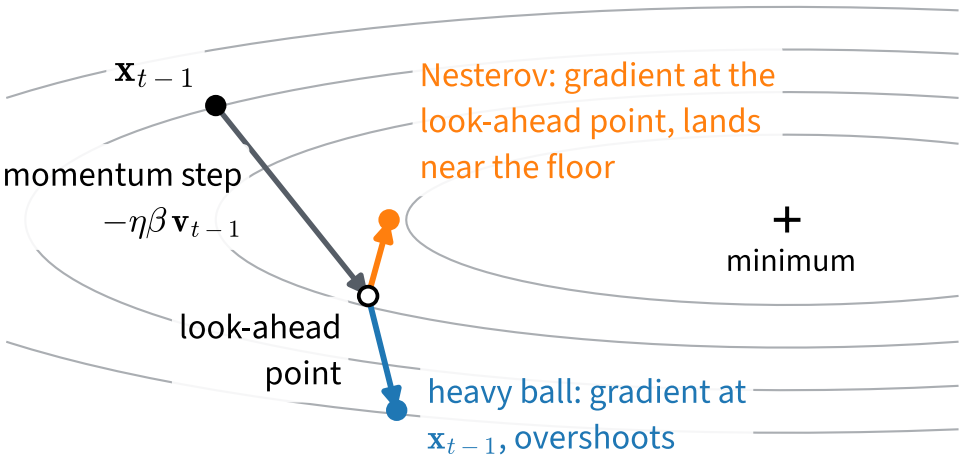
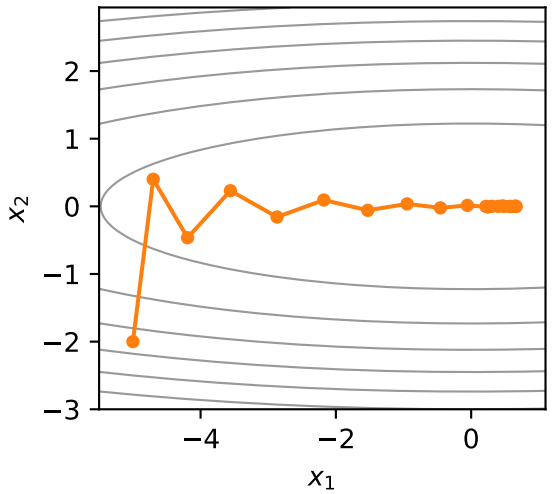


Figure 9.7: One step of heavy-ball and of Nesterov momentum on the valley, from the same iterate and the same velocity. Both first move by the momentum step to the look-ahead point. Heavy ball then adds the gradient step computed at the old iterate and overshoots across the valley; Nesterov adds the gradient step computed at the look-ahead point, which points back, and lands near the floor.

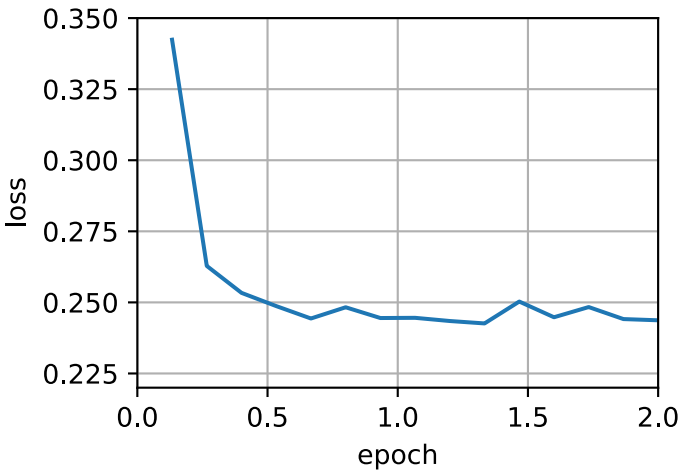


At the same $\eta = 0.3$ and $\beta = 0.8$ as the underdamped heavy-ball run, the zigzag across the valley dies out after the first steps, and the overshoot along the floor is smaller. The look-ahead acts as extra damping, strongest in the steep direction, where the overshoot is largest. Nesterov's method also has guarantees that heavy-ball momentum lacks. On smooth convex functions, with a momentum that grows toward 1 over the steps, its error after k steps falls as $O(1/k^2)$, against $O(1/k)$ for gradient descent, and no method built from gradients does better in the worst case. On strongly convex functions a constant momentum $\beta = (\sqrt{\kappa} - 1)/(\sqrt{\kappa} + 1)$ achieves the $\sqrt{\kappa}$ rate, with a proof that is not confined to quadratics (Nesterov 2018). Section 35.1.3 states the result, proves it on quadratics, and cites the general proof.

The SGD optimizer of each framework offers Nesterov momentum as the flag `nesterov=True`, which is off by default. The implementation is a rewrite of Equation 9.25 that takes the gradient at the parameters it stores, so the look-ahead needs no extra gradient evaluation. Exercise 3 works out how these parameters relate to the iterates of Equation 9.25.

```
d2l.train_with_optimizer(
    torch.optim.SGD,
    {'lr': 0.005, 'momentum': 0.9, 'nesterov': True}, data_iter);
```

```
loss: 0.244, 0.503 sec/epoch
```



On the airfoil data the loss curve is nearly the same as with plain momentum. At a batch size of 10 the sampling noise is much larger than the correction from the look-ahead. The correction matters when curvature rather than noise limits progress, as in full-batch or large-batch training with β close to 1.

Momentum fixes the slow progress along the flat directions of a valley, but it does so with one learning rate for all coordinates, and that rate is still capped by the steepest direction. The next section gives each coordinate a step size of its own.

9.4.5 Summary

Momentum replaces the gradient with a leaky average of past gradients, the velocity, using one buffer per parameter and one hyperparameter β. On an ill-conditioned valley the components of the gradient that persist from step to step accumulate up to $1/(1-\beta)$-fold, while those that alternate cancel. With tuned η and β the number of steps falls from order κ to order $\sqrt{\kappa}$: on the valley with $\kappa = 20$, 40 steps instead of 139 to reach 10^{-6}. The parameter β is a timescale, an average over about $1/(1-\beta)$ gradients, and a damping knob: past critical damping the iterate rings around the minimum. The effective step is $\eta/(1-\beta)$, so raising β requires lowering η. Nesterov's look-ahead damps the ringing and carries guarantees beyond quadratics without extra work per step. What momentum cannot do is give different coordinates different step sizes.

9.4.6 Exercises

1. code **Momentum and learning rate together.** On the valley `f_valley` with `momentum_2d`, run a grid of $\beta \in \{0, 0.25, 0.5, 0.8\}$ and $\eta \in \{0.1, 0.3, 0.6, 1.0\}$ for up to 1000 steps each.
    1. Record for every cell the number of steps until $\|\mathbf{x}_t\| \le 10^{-3}$, or the fact that the iterates diverged, and display the grid as a table or heatmap.
    2. Compare the fastest β at each stable η with the $\beta^\star$ of Equation 9.24 for $\kappa = 20$, and explain each learning rate at which they differ.
2. **Velocity noise.** Let $\mathbf{g}_t = \mathbf{g} + \xi_t$, where the noise ξ_t is independent across steps, has zero mean, and has variance σ^2 per coordinate. Show that the stationary mean of $\mathbf{v}_t$ in Equation 9.22 is $\mathbf{g}/(1-\beta)$ and its stationary variance is $\sigma^2/(1-\beta^2)$. Conclude that the noise-to-signal ratio of the velocity is $\sqrt{(1-\beta)/(1+\beta)}$ times that of a single gradient.
3. **Two forms of Nesterov momentum.** Library implementations of Nesterov momentum store a parameter vector $\mathbf{y}_t$ and evaluate the gradient there, $\mathbf{g}_t = \nabla f(\mathbf{y}_{t-1})$. Starting from $\mathbf{y}_0 = \mathbf{x}_0$ and $\mathbf{v}_0 = \mathbf{0}$, they update
$$\mathbf{v}_t = \beta \mathbf{v}_{t-1} + \mathbf{g}_t, \qquad \mathbf{y}_t = \mathbf{y}_{t-1} - \eta\,(\mathbf{g}_t + \beta \mathbf{v}_t),$$
so that no extra gradient evaluation is needed.
    1. Express $\mathbf{y}_t$ through the $\mathbf{x}_t$ and $\mathbf{v}_t$ of Equation 9.25 so that both recurrences produce the same velocities, and prove the relation by induction over t.
    2. A checkpoint saves the library's parameters $\mathbf{y}_t$. Which point of Equation 9.25 does it store, and what else must be saved to recover $\mathbf{x}_t$?
4. **Stability of momentum.** ▣ The effective-step reading of Section 9.4.2.3 suggests that raising β forces η down. Does the largest stable η in the grid of problem 1 behave that way? Derive the stability condition of Equation 9.22 for a single eigendirection with curvature λ and reconcile the two.
5. code **Many eigenvalues.** ▣ Consider $f(\mathbf{x}) = \frac{1}{2}\sum_{i=1}^{10} \lambda_i x_i^2$ with $\lambda_i = 2^{-i}$, started from $x_i = 1$, and let $\kappa = \lambda_{\max}/\lambda_{\min}$.
    1. Run gradient descent at $\eta = 2/(\lambda_{\max} + \lambda_{\min})$, the learning rate that minimizes its worst-case contraction factor $\max_i |1 - \eta\lambda_i|$. Run heavy-ball momentum at the $\eta^\star$ and $\beta^\star$ of Equation 9.24. Plot $|x_{i,t}|$ for every coordinate on a logarithmic scale.
    2. Which coordinates limit the convergence of each method? Compare the ratio of the iteration counts until $\|\mathbf{x}_t\| \le 10^{-6}$ with $\sqrt{\kappa}$.
6. code **Momentum under minibatch noise.** ▣ Keep the noise model of problem 2, but let the gradient depend on the iterate: on $f(x) = \frac{\lambda}{2}x^2$, take $g_t = \lambda x_{t-1} + \xi_t$.
    1. Compare plain SGD at learning rate α with momentum $\beta = 0.9$ at $\eta = \alpha(1-\beta)$, which keeps the effective step $\eta/(1-\beta)$ equal to α. Compute the ratio of the two stationary variances of x_t for $\alpha\lambda \in \{0.05, 0.2\}$, by simulation or by solving for the stationary covariance of (x_t, v_t). Does the reduction of the velocity's noise-to-signal ratio from problem 2 carry over to the iterate? Explain the result.

2. On the airfoil data, rebuild `data_iter` and `feature_dim` with `d2l.get_airfoil(batch_size=b)` for $b \in \{1, 10, 100\}$. Run `d2l.train_with_rule` with `sgd_momentum` and `num_epochs=10` at $\beta \in \{0, 0.9\}$, choosing η so that $\eta/(1-\beta) = 0.05$. Report the final loss of each run and the spread of the recorded losses over its last epoch, and compare the effect of momentum with the prediction of the first item.

9.5 Adam and AdamW

The modern training recipe of Section 7.6 replaced SGD with momentum by AdamW, and we used it there as one ingredient among many. Adam differs from the optimizers so far in one respect: it gives every coordinate of the parameter vector its own learning rate. This section asks what such per-coordinate rates achieve, how Adam sets them without knowing the curvature, and why weight decay must then bypass them.

The valley $f(\mathbf{x}) = 0.1x_1^2 + 2x_2^2$ of Section 9.2 shows what a single rate costs. Gradient descent uses one learning rate for every coordinate, and the stiffest direction decides how large that rate may be. Here the curvatures are 0.2 and 4, so a step is stable only for $\eta < 2/4 = 0.5$. At $\eta = 0.4$ the flat coordinate shrinks by a factor of only $1 - 0.4 \cdot 0.2 = 0.92$ per step, and twenty steps take x_1 from -5 to about -0.94. With a separate rate for each coordinate, the inverse of that coordinate's curvature, a single step would reach the minimum. Momentum averages gradients over time and so eases the problem, but it keeps the single rate.

The curvatures of a neural network are unknown, and they change during training. The methods of this section therefore estimate a scale for each coordinate from the history of that coordinate's own gradients. AdaGrad accumulates the squared gradients, RMSProp lets the accumulation forget, and Adam (Kingma and Ba 2015) adds momentum and corrects both averages for their start at zero. Adam is the standard optimizer for transformers. To see where its advantage comes from, we build two small models, a transformer language model and a convolutional network, and compare Adam with tuned SGD with momentum on both. The section ends with AdamW, the form in which Adam is used in practice. AdamW applies weight decay outside the per-coordinate scaling. It also stores two extra numbers for every parameter, and we count that memory at the end.

```
import copy
import os
import pandas as pd
import random
import textwrap
from functools import partial
```

9.5.1 Per-Coordinate Learning Rates

9.5.1.1 AdaGrad

Consider learning problems whose features occur with very different frequencies. This is the normal state of affairs in language: we will see the word *learning* far more often than the word *preconditioning*. The same imbalance appears in advertising and recommendation, where most items interest only a few users. Parameters tied to rare features receive informative gradients only when those features occur. Under the decaying learning rate

that stochastic convergence demands (Section 9.3), the parameters of frequent features approach their optima quickly while the rare ones are still waiting for evidence, and by the time it arrives the shared step size has decayed too far to use it. One clock cannot serve both populations.

AdaGrad (Duchi et al. 2011) gives every coordinate its own clock. Instead of decaying the step with the iteration count t, it decays each coordinate's step with that coordinate's own activity, measured by the running sum of its squared gradients:

$$\mathbf{s}_t = \mathbf{s}_{t-1} + \mathbf{g}_t^2, \qquad \mathbf{x}_{t+1} = \mathbf{x}_t - \frac{\eta}{\sqrt{\mathbf{s}_t + \epsilon}} \odot \mathbf{g}_t, \tag{9.26}$$

with all operations elementwise, $\mathbf{s}_0 = \mathbf{0}$, and a small $\epsilon > 0$ guarding the division. A coordinate whose gradients have been large moves cautiously from now on. A coordinate that has barely been touched keeps a large step and can respond as soon as its rare feature appears.

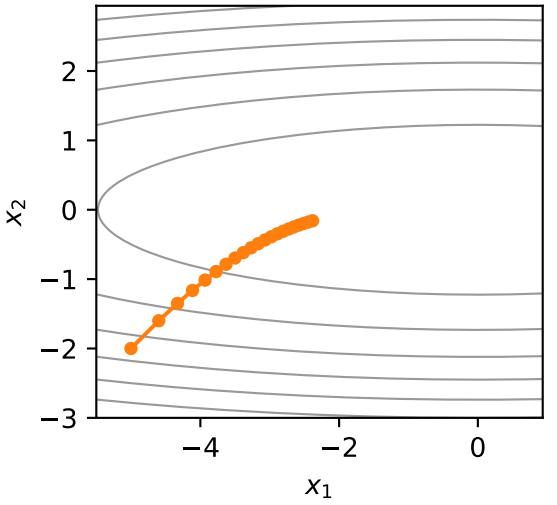
The same rule addresses the valley. The ideal fix there divides the step in each direction by the curvature of that direction. A transformation that rescales the directions of a problem in this way is called a *preconditioner*. Curvature is too expensive to compute at the scale of deep learning. When the coordinate axes roughly align with the directions of curvature, however, the size of a coordinate's past gradients is a usable stand-in for its curvature. Then Equation 9.26 is a diagonal preconditioner estimated from gradients alone. Section 35.5.1 makes both readings precise, through a change of metric and through a regret bound.

We run AdaGrad on the valley from the starting point $(-5, -2)$ at the learning rate $\eta = 0.4$, close to the stability limit of gradient descent. The helper `d2l.train_2d` of Section 9.2 carries the state (s_1, s_2) from step to step and passes extra keyword arguments, such as `eta`, to the update rule; `d2l.f_valley` and `d2l.valley_grad` are the valley and its gradient.

```
def adagrad_2d(x1, x2, s1, s2, f_grad, eta=0.4, eps=1e-6):
    g1, g2 = f_grad(x1, x2)
    s1, s2 = s1 + g1 ** 2, s2 + g2 ** 2
    x1 -= eta / math.sqrt(s1 + eps) * g1
    x2 -= eta / math.sqrt(s2 + eps) * g2
    return x1, x2, s1, s2

d2l.show_trace_2d(d2l.f_valley, d2l.train_2d(adagrad_2d, d2l.valley_grad))
```

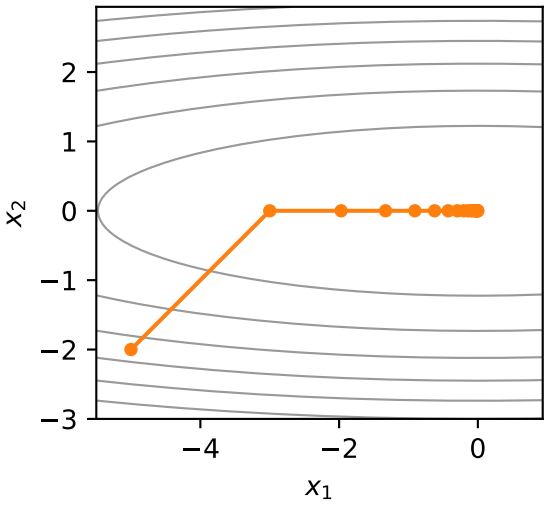
```
step 20, x1: -2.382563, x2: -0.158591
```



The path is smooth: the steep direction x_2 is scaled down automatically, so it does not oscillate. But the accumulation also has a drawback. On the first step $\mathbf{s}_1 = \mathbf{g}_1^2$, so every coordinate moves by exactly η, whatever its gradient. After that, a coordinate whose gradient stays roughly the same size moves by about $\eta/\sqrt{t}$ at step t, and twenty such steps add up to at most $\eta \sum_{t=1}^{20} t^{-1/2} \approx 7.6\,\eta = 3.0$. That is not enough to cover the distance of 5 along x_1, and the iterate stalls far from the minimum. Because the scaling adapts, we can compensate with a learning rate at which gradient descent diverges, $\eta = 2$:

```
d2l.show_trace_2d(d2l.f_valley, d2l.train_2d(adagrad_2d, d2l.valley_grad,
                                             eta=2))
```

```
step 20, x1: -0.002295, x2: -0.000000
```



9.5.1.2 RMSProp

AdaGrad's sum never forgets. Under persistent gradient noise $\mathbf{s}_t$ grows roughly linearly in t, so the effective step decays like $t^{-1/2}$ whether or not the optimization is anywhere near done. This decay suits the convex problems that AdaGrad was designed for, but it can stop a nonconvex optimization that still needs to move late in training.

RMSProp (Tieleman and Hinton 2012) keeps the per-coordinate scaling and replaces the sum with an exponential moving average, the same leaky averaging that momentum applies to the gradient itself:

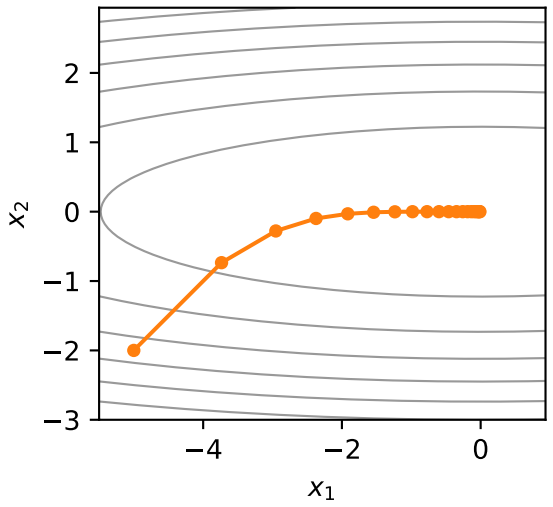
$$\mathbf{v}_t = \beta_2 \mathbf{v}_{t-1} + (1-\beta_2)\,\mathbf{g}_t^2, \qquad \mathbf{x}_{t+1} = \mathbf{x}_t - \frac{\eta}{\sqrt{\mathbf{v}_t + \epsilon}} \odot \mathbf{g}_t. \tag{9.27}$$

The average $\mathbf{v}_t$ estimates the *current* squared-gradient scale rather than the lifetime total, over a window of about $1/(1-\beta_2)$ steps, ten at the customary $\beta_2 = 0.9$. The step size no longer decays by construction, and the learning rate η becomes a setting we control separately, to be changed over time by a schedule (Section 9.6). On the valley at $\eta = 0.4$, RMSProp gets close to the minimum where AdaGrad stalled:

```
def rmsprop_2d(x1, x2, v1, v2, f_grad, eta=0.4, beta2=0.9, eps=1e-6):
    g1, g2 = f_grad(x1, x2)
    v1 = beta2 * v1 + (1 - beta2) * g1 ** 2
    v2 = beta2 * v2 + (1 - beta2) * g2 ** 2
    x1 -= eta / math.sqrt(v1 + eps) * g1
    x2 -= eta / math.sqrt(v2 + eps) * g2
    return x1, x2, v1, v2

d2l.show_trace_2d(d2l.f_valley, d2l.train_2d(rmsprop_2d, d2l.valley_grad))
```

```
step 20, x1: -0.010599, x2: 0.000000
```



The first step of this run is larger than intended. Since $\mathbf{v}_0 = \mathbf{0}$, the average after one step is $\mathbf{v}_1 = 0.1\,\mathbf{g}_1^2$, a tenth of the squared gradient, and each coordinate moves by $\eta/\sqrt{0.1} \approx 3.2\,\eta$

instead of η. The average starts at zero and needs about ten steps to fill its window. Adam corrects this startup error exactly.

9.5.1.3 Adam

Adam applies the same exponential averaging to the gradient itself, giving a momentum estimate $\mathbf{m}_t$, keeps RMSProp's second-moment estimate $\mathbf{v}_t$, and divides each average by the total weight it has accumulated so far:

$$\begin{aligned} \mathbf{m}_t &= \beta_1\,\mathbf{m}_{t-1} + (1-\beta_1)\,\mathbf{g}_t, & \hat{\mathbf{m}}_t &= \mathbf{m}_t/(1-\beta_1^t), \\ \mathbf{v}_t &= \beta_2\,\mathbf{v}_{t-1} + (1-\beta_2)\,\mathbf{g}_t^2, & \hat{\mathbf{v}}_t &= \mathbf{v}_t/(1-\beta_2^t). \end{aligned} \tag{9.28}$$

The update divides the corrected momentum by the square root of the corrected scale:

$$\mathbf{x}_{t+1} = \mathbf{x}_t - \eta\,\frac{\hat{\mathbf{m}}_t}{\sqrt{\hat{\mathbf{v}}_t} + \epsilon}. \tag{9.29}$$

Each coordinate thus moves by about η times the ratio of its average gradient to its typical gradient size. The momentum method of Section 9.4 *adds* each gradient to β times its buffer. The factor $1-\beta_1$ turns that sum into an *average*, which amounts to the same method with the learning rate multiplied by $1-\beta_1$. The defaults are $\beta_1 = 0.9$ and $\beta_2 = 0.999$: the direction estimate averages over about ten steps, the scale estimate over about a thousand. Language-model recipes since GPT-3 (Brown et al. 2020) often use $\beta_2 = 0.95$, a window of twenty steps. The scale estimate then follows changes in the gradients more quickly, but with more noise.

The corrections in Equation 9.28 are exact rather than heuristic. Unrolled from $\mathbf{v}_0 = \mathbf{0}$, the average $\mathbf{v}_t = (1-\beta_2)\sum_{s=1}^{t}\beta_2^{t-s}\,\mathbf{g}_s^2$ is a weighted sum whose weights total $1-\beta_2^t$, not 1. If the squared gradients have a steady mean, $\mathbf{v}_t$ therefore carries the fraction $1-\beta_2^t$ of it in expectation, and dividing by that fraction removes the deficit at every t. The deficit is large: at $\beta_2 = 0.999$, after ten steps $\mathbf{v}_t$ holds about 1% of its stationary value. Without the corrections, the first step would be $(1-\beta_1)/\sqrt{1-\beta_2} = 0.1/\sqrt{0.001} \approx 3.2$ times η, the same overshoot as RMSProp's. With them, $\hat{\mathbf{m}}_1 = \mathbf{g}_1$ and $\hat{\mathbf{v}}_1 = \mathbf{g}_1^2$, so the first step moves every coordinate by exactly η. Section 35.5.1 plots the transient and its cancellation.

Adam also adds ϵ *outside* the square root, unlike Equation 9.27. Where $\sqrt{\hat{\mathbf{v}}_t}$ is of order one the placement matters little. It decides the update where $\sqrt{\hat{\mathbf{v}}_t}$ is small, and we return to it when we look at the limits of the method. The usual default is $\epsilon = 10^{-8}$.

9.5.1.4 Adam from Scratch

The update needs two buffers per parameter and a step counter for the bias correction. We write it the way the framework writes its own optimizers, so that it can train any model.

```
class Adam(torch.optim.Optimizer):
    """Adam from scratch: steps scaled by two debiased moving averages."""
    def __init__(self, params, lr=1e-3, betas=(0.9, 0.999), eps=1e-8):
        super().__init__(params, dict(lr=lr, betas=betas, eps=eps))
```

```
    @torch.no_grad()
    def step(self):
        for group in self.param_groups:
            (beta1, beta2), lr, eps = group['betas'], group['lr'], group['eps']
            for p in group['params']:
                if p.grad is None:  # A parameter that received no gradient
                    continue
                state = self.state[p]
                if not state:  # First step: t = 0 and m = v = 0
                    state.update(t=0, m=torch.zeros_like(p),
                                 v=torch.zeros_like(p))
                state['t'] += 1
                t, m, v = state['t'], state['m'], state['v']
                m.mul_(beta1).add_(p.grad, alpha=1 - beta1)
                v.mul_(beta2).add_(p.grad ** 2, alpha=1 - beta2)
                m_hat, v_hat = m / (1 - beta1 ** t), v / (1 - beta2 ** t)
                p.sub_(lr * m_hat / (v_hat.sqrt() + eps))
```

Before using our version we check it against the library on a small least-squares problem: two copies of the same linear model, the same data, 100 steps each. The first step should move every weight by exactly η.

```
def compare(ours, theirs, lr, num_steps=100):
    X, y = torch.randn(512, 20), torch.randn(512, 1)
    net = nn.Linear(20, 1)
    twin = copy.deepcopy(net)
    optimizers = [ours(net.parameters(), lr), theirs(twin.parameters(), lr)]
    w0 = net.weight.detach().clone()
    for t in range(num_steps):
        for model, optimizer in zip([net, twin], optimizers):
            loss = F.mse_loss(model(X), y)
            optimizer.zero_grad()
            loss.backward()
            optimizer.step()
        if t == 0:
            first = (net.weight - w0).abs() / lr
            print(f'first step / lr: {first.min():.4f} to {first.max():.4f}')
    print(f'largest difference after {num_steps} steps: '
          f'{(net.weight - twin.weight).abs().max():.1e}')

compare(Adam, torch.optim.Adam, lr=0.01)
```

```
first step / lr: 1.0000 to 1.0000
largest difference after 100 steps: 1.5e-08
```

The first step moves every weight by η, as the bias correction promises, and after 100 steps the two copies agree to within rounding error. From now on we call the library version.

9.5.2 TinyLM and FashionCNN

A comparison of optimizers needs models on which the optimizers behave differently, and models small enough to train many times. We use two of about the same size: a transformer language model, the kind of model Adam is used for in practice, and a convolutional image classifier, the kind of model that SGD with momentum trained for a decade. Both go into the `d2l` library, and the rest of this chapter trains them.

9.5.2.1 The Language Model

The data is WikiText-103 (Merity et al. 2016), about 100 million words of Wikipedia articles kept as raw text, with case, digits, and punctuation. A run of 2,000 steps on minibatches of 64 windows of 64 tokens makes about 8 million predictions. On the 100,000 characters of *The Time Machine* each character would be a target about 80 times, and the model would memorize the text. Here the training windows start at random offsets in the first of the two training files, about 270 million bytes, so a run reads about 3% of it and rarely sees the same text twice. The tokens are the UTF-8 bytes of the text: the byte-level tokenizer of Section 8.2 without any merges, a vocabulary of 256 tokens, so the loss converts directly to bits per byte (Section 8.3). Validation windows are consecutive, non-overlapping pieces of the official validation split, about 1.1 million bytes.

```
class WikiText103(d2l.DataModule):
    """WikiText-103 as UTF-8 bytes, cut into windows of num_steps bytes."""
    def __init__(self, batch_size, num_steps, num_train=None, num_val=None):
        super().__init__()
        self.save_hyperparameters()
        self.vocab = d2l.BPETokenizer(vocab_size=256, specials=())  # Bytes
        self.train = self._read('train-00000-of-00002',
                                '3cbc792dfc0eeedf8d7333acdc15a9f536032d9a')
        self.val = self._read('validation-00000-of-00001',
                              '49d15019b0476afc752bd5f010b5a2aab2de89ca')
        self.train, self.val = self.train[:num_train], self.val[:num_val]

    def _read(self, split, sha1):
        url = ('https://huggingface.co/datasets/Salesforce/wikitext/resolve/'
               f'main/wikitext-103-raw-v1/{split}.parquet')
        fname = d2l.download(url, os.path.join(self.root, 'wikitext-103'),
                             sha1)
        lines = pd.read_parquet(fname)['text']  # Each line ends with '\n'
        return np.frombuffer(b''.join(s.encode() for s in lines), np.uint8)

    def windows(self, corpus, starts):
        """Inputs and next-byte targets of the windows at `starts`."""
        idx = np.asarray(starts)[:, None] + np.arange(self.num_steps + 1)
        array = torch.from_numpy(corpus[idx].astype(np.int64))
        return array[:, :-1], array[:, 1:]

    def train_dataloader(self):  # A pass of minibatches at random offsets
        n = len(self.train) - self.num_steps
        for _ in range(n // (self.batch_size * self.num_steps)):
            yield self.windows(self.train,
                               torch.randint(n, (self.batch_size,)).numpy())
```

```
    def val_dataloader(self):  # Consecutive, non-overlapping windows
        starts = np.arange(0, len(self.val) - self.num_steps, self.num_steps)
        for i in range(0, len(starts), self.batch_size):
            yield self.windows(self.val, starts[i:i + self.batch_size])
```

Both splits are held in memory as arrays of bytes, one byte per token, and a minibatch is gathered from them on demand. A training pass draws as many windows as the text holds non-overlapping ones, each at an independent random offset, so `train_steps` below can run for any number of steps without reusing a fixed set of windows.

```
data = WikiText103(batch_size=64, num_steps=64)
print(f'{len(data.train):,} training bytes, '
      f'{len(data.val):,} validation bytes')
text = data.vocab.decode(data.train[:300])
print('\n'.join(textwrap.fill(line, 72) for line in text.splitlines()))
```

```
269,630,498 training bytes, 1,144,248 validation bytes
 = Valkyria Chronicles III =
 Senjō no Valkyria 3 : Unrecorded Chronicles ( Japanese :           3 ,
lit . Valkyria of the Battlefield 3 ) , commonly referred to as Valkyria
Chronicles III outside Japan , is a tactical role @-@ playing video game
developed by Sega and Media.Vis
```

The model, `TinyLM`, is a *decoder-only transformer*, the architecture of Chapter 11. We use it two chapters early because an optimizer does not need to know what attention does: it sees a differentiable function with parameters of several kinds, and only their gradients. The model has learned token and position embeddings, two blocks, and a linear head that predicts the next byte. Each block has an attention layer and a two-layer MLP, each behind a LayerNorm and each with a residual connection. The attention computation is a single library call. Every linear layer draws its weights and biases uniformly from $\pm 1/\sqrt{d_{\text{in}}}$, and every embedding row from a standard normal distribution.

```
class TinyLM(nn.Module):
    """A small decoder-only transformer language model."""
    def __init__(self, vocab_size, d_model=128, num_heads=2, num_blks=2,
                 max_len=64):
        super().__init__()
        self.num_heads = num_heads
        self.token_emb = nn.Embedding(vocab_size, d_model)
        self.pos_emb = nn.Embedding(max_len, d_model)
        self.blks = nn.ModuleList([nn.ModuleDict(dict(
            norm1=nn.LayerNorm(d_model),
            qkv=nn.Linear(d_model, 3 * d_model),
            proj=nn.Linear(d_model, d_model),
            norm2=nn.LayerNorm(d_model),
            mlp=nn.Sequential(nn.Linear(d_model, 4 * d_model), nn.GELU(),
                              nn.Linear(4 * d_model, d_model))))
            for _ in range(num_blks)])
        self.norm = nn.LayerNorm(d_model)
        self.head = nn.Linear(d_model, vocab_size)
```

```
    def attention(self, blk, X):
        B, T, D = X.shape
        q, k, v = blk['qkv'](X).chunk(3, dim=-1)
        q, k, v = (u.reshape(B, T, self.num_heads, -1).transpose(1, 2)
                   for u in (q, k, v))
        Y = F.scaled_dot_product_attention(q, k, v, is_causal=True)
        return blk['proj'](Y.transpose(1, 2).reshape(B, T, D))

    def forward(self, X):
        H = self.token_emb(X) + self.pos_emb(torch.arange(X.shape[1],
                                                          device=X.device))
        for blk in self.blks:
            H = H + self.attention(blk, blk['norm1'](H))
            H = H + blk['mlp'](blk['norm2'](H))
        return self.head(self.norm(H))
```

The initialization described above is the default of `nn.Linear` and `nn.Embedding`, and `nn.GELU` computes the exact GELU.

9.5.2.2 Parameter Groups

The parameters of `TinyLM` fall into three groups. *Embeddings* are tables whose rows receive a gradient only when their token or position occurs, the sparse features that motivated AdaGrad. Two-dimensional *matrices*, the weights of the linear layers, hold nearly all of the parameters. One-dimensional *vectors* are the biases and the LayerNorm scales and shifts. Weight decay at the end of this section and Muon in Section 9.7 treat the three groups differently, so we count them once here.

```
model = TinyLM(vocab_size=len(data.vocab))
kinds = {'embeddings': 0, 'matrices': 0, 'vectors': 0}
print(f'{"parameter":<22} {"shape":<14} {"count":>7}')
for name, p in model.named_parameters():
    kind = ('embeddings' if 'emb' in name else
            'matrices' if p.ndim == 2 else 'vectors')
    kinds[kind] += p.numel()
    print(f'{name:<22} {str(tuple(p.shape)):<14} {p.numel():>7}')
for kind, count in kinds.items():
    print(f'{kind:>37} {count:>7}')
print(f'{"total":>37} {sum(kinds.values()):>7}')
```

```
parameter              shape           count
token_emb.weight       (256, 128)      32768
pos_emb.weight         (64, 128)        8192
blks.0.norm1.weight    (128,)            128
blks.0.norm1.bias      (128,)            128
blks.0.qkv.weight      (384, 128)      49152
blks.0.qkv.bias        (384,)            384
blks.0.proj.weight     (128, 128)      16384
blks.0.proj.bias       (128,)            128
blks.0.norm2.weight    (128,)            128
blks.0.norm2.bias      (128,)            128
```

```
blks.0.mlp.0.weight    (512, 128)       65536
blks.0.mlp.0.bias      (512,)             512
blks.0.mlp.2.weight    (128, 512)       65536
blks.0.mlp.2.bias      (128,)             128
blks.1.norm1.weight    (128,)             128
blks.1.norm1.bias      (128,)             128
blks.1.qkv.weight      (384, 128)       49152
blks.1.qkv.bias        (384,)             384
blks.1.proj.weight     (128, 128)       16384
blks.1.proj.bias       (128,)             128
blks.1.norm2.weight    (128,)             128
blks.1.norm2.bias      (128,)             128
blks.1.mlp.0.weight    (512, 128)       65536
blks.1.mlp.0.bias      (512,)             512
blks.1.mlp.2.weight    (128, 512)       65536
blks.1.mlp.2.bias      (128,)             128
norm.weight            (128,)             128
norm.bias              (128,)             128
head.weight            (256, 128)       32768
head.bias              (256,)             256
                    embeddings   40960
                      matrices  425984
                       vectors    3840
                         total  470784
```

The model has about 0.47 million parameters, 90% of them in the matrices.

9.5.2.3 Training for a Fixed Number of Steps

Training is a loop counted in steps rather than epochs: draw a minibatch, compute the mean cross-entropy of the next-token predictions, take an optimizer step, record the loss. Nothing in it is specific to language: it trains any model that maps a minibatch to logits, so we save it for the rest of the book as `train_steps`. Positions labeled -100 add no loss, which lets a model train on part of its outputs only. An optional function `stop` of the step count ends a run early once it returns true.

```
def train_steps(model, data, optimizer, num_steps, *, stop=None):
    """Minimize the mean cross-entropy of the logits model(X) for the labels
    Y over the minibatches (X, Y) of data, leaving out the labels -100, for
    num_steps steps or until stop(step) is true; return every step's loss."""
    device = d2l.try_gpu()
    model.to(device)
    losses, step = [], 0
    while step < num_steps:
        for X, Y in data.train_dataloader():
            X, Y = X.to(device), Y.to(device)
            logits = model(X)
            loss = F.cross_entropy(logits.reshape(-1, logits.shape[-1]),
                                   Y.reshape(-1), ignore_index=-100)
            optimizer.zero_grad()
            loss.backward()
            optimizer.step()
            losses.append(loss.detach())
```

```
            step += 1
            if step >= num_steps or (stop is not None and stop(step)):
                return torch.stack(losses).tolist()
```

The counterpart for evaluation is `lm_perplexity`. It sums the cross-entropy of the next-token predictions over a set of minibatches, usually the validation set, with dropout off, and returns the exponential of the mean, the perplexity of Section 8.3. Its logarithm is the mean loss per token.

```
def lm_perplexity(model, batches):
    """Perplexity of a language model on the minibatches (X, Y) of batches."""
    device = next(model.parameters()).device
    modes = {m: m.training for m in model.modules()}  # Restored at the end
    model.eval()  # Dropout off while we measure
    total, count = 0, 0
    with torch.inference_mode():
        for X, Y in batches:
            logits = model(X.to(device))
            total += F.cross_entropy(logits.flatten(0, 1),
                                     Y.to(device).flatten(), reduction='sum')
            count += Y.numel()
    for m, training in modes.items():
        m.training = training
    return math.exp(float(total) / count)
```

Three helpers read a run: the final loss, averaged over the last hundred steps and set to infinity if the run diverged; a curve smoothed over blocks of 25 steps for plotting; and the first step at which the smoothed curve reaches a given loss, or `None` if it never does. Later sections use the first two as well.

```
def final_loss(losses, k=100):
    """Mean of the last k losses; infinity if the run diverged."""
    v = sum(losses[-k:]) / k
    return v if math.isfinite(v) else float('inf')

def smooth(losses, k=25):
    """Means over consecutive blocks of k losses, for plotting."""
    return [sum(losses[i:i + k]) / k
            for i in range(0, len(losses) - k + 1, k)]

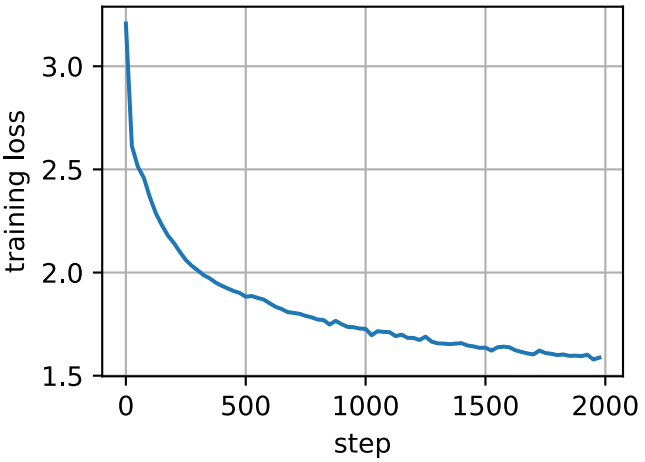
def steps_to_reach(losses, target, k=25):
    for i, v in enumerate(smooth(losses, k)):
        if v <= target:
            return k * (i + 1)
    return None
```

A first run trains `TinyLM` with Adam at $\eta = 0.003$ for 2,000 steps:

```
model = TinyLM(len(data.vocab))
optimizer = torch.optim.Adam(model.parameters(), lr=0.003)
```

```
losses = train_steps(model, data, optimizer, num_steps=2000)
d2l.plot(list(range(0, 2000, 25)), [smooth(losses)],
         'step', 'training loss')
print(f'training perplexity per byte: '
      f'{math.exp(final_loss(losses)):.2f}')
```

```
training perplexity per byte: 4.91
```



The model reaches a training perplexity per byte of about 5, or about 2.3 bits per byte, against 256 for a uniform guess over the bytes: it has learned a good deal of English spelling, and its loss is still falling slowly when the run ends. Strong language models compress text like this to below one bit per byte (Section 8.3).

9.5.2.4 The Image Classifier

`FashionCNN` classifies the 28×28 grayscale images of Fashion-MNIST (Section 3.2). Two stages of a 3×3 convolution, a ReLU, and 2×2 max-pooling, with 32 and 64 channels, reduce each image to $64 \times 7 \times 7$ features. Two fully connected layers map them to 128 hidden units and then to the ten classes. Every weight and bias is drawn uniformly from $\pm 1/\sqrt{\text{fan-in}}$, where the fan-in of a unit is the number of inputs it sums over.

```
class FashionCNN(nn.Module):
    """A compact CNN for Fashion-MNIST: two conv-pool stages, two linear."""
    def __init__(self):
        super().__init__()
        self.conv1 = nn.Conv2d(1, 32, kernel_size=3, padding=1)
        self.conv2 = nn.Conv2d(32, 64, kernel_size=3, padding=1)
        self.fc1 = nn.Linear(64 * 7 * 7, 128)
        self.fc2 = nn.Linear(128, 10)

    def forward(self, X):
        X = F.max_pool2d(F.relu(self.conv1(X)), 2)
        X = F.max_pool2d(F.relu(self.conv2(X)), 2)
        return self.fc2(F.relu(self.fc1(X.flatten(1))))
```

The initialization described above is the default of `nn.Conv2d` and `nn.Linear`. Images arrive as (batch, channel, height, width).

With 421,642 parameters, `FashionCNN` is about the same size as `TinyLM`. Since `train_steps` only asks a model for logits, it trains the CNN unchanged; Fashion-MNIST minibatches of 256 images take the place of the byte windows. The CNN also needs a test accuracy: `evaluate_accuracy` returns the fraction of the examples in a set of minibatches that a model classifies correctly.

```
fashion = d2l.FashionMNIST(batch_size=256)

def evaluate_accuracy(model, batches):
    """Fraction of the examples in batches that model classifies correctly."""
    device = next(model.parameters()).device
    training, correct, total = model.training, 0, 0
    model.eval()  # Dropout off, batch norm on running statistics
    with torch.inference_mode():
        for X, y in batches:
            y_hat = model(X.to(device)).argmax(dim=1)
            correct += (y_hat == y.to(device)).sum()
            total += len(y)
    model.train(training)
    return float(correct) / total
```

9.5.3 Strengths and Limits of Adam

Practitioners train transformers with Adam because SGD, however carefully tuned, falls behind. On convolutional networks the case for Adam is far weaker, and SGD with momentum was the standard for a decade of computer vision. Claims that one optimizer beats another depend strongly on how much effort went into tuning each (Schmidt et al. 2021), so the protocol of the comparison matters as much as its result.

Ours is symmetric and small. Both optimizers train the same model from the same initialization, a random draw shared by every run of the comparison, for 2,000 steps at a constant learning rate, with no schedule, no clipping, and no weight decay. Each gets a grid of four learning rates spaced by factors of about three, and the run with the lowest final training loss represents its method. Training loss is the right measure here, because the question is which method optimizes faster, not which one generalizes better. For the language model the two would agree anyway, since a run rarely sees the same text twice. Below, "tuned" means the best of these four rates.

9.5.3.1 Comparison on the Language Model

The function `lr_sweep` trains one fresh model per learning rate, every one from the same initialization, and prints its final loss unless `verbose` is false. Given a function `evaluate`, it also evaluates each model, at the end of training or every `every` steps, and prints the last value under the name `metric`. Later sections run their comparisons with it as well.

```
def lr_sweep(make_model, make_optimizer, data, lrs, seed,
             num_steps=2000, *, evaluate=None, metric='score', every=None,
```

```
                verbose=True):
    """Train make_model() with make_optimizer(model, lr) for each lr, from
    one initialization; return the loss curves and the evaluations."""
    every = every or num_steps
    curves, scores = {}, {}
    for lr in lrs:
        torch.manual_seed(seed)  # The same initialization for every lr
        model = make_model()
        optimizer = make_optimizer(model, lr)
        curves[lr], scores[lr] = [], []
        for _ in range(num_steps // every):
            curves[lr] += train_steps(model, data, optimizer, every)
            if evaluate is not None:
                scores[lr].append(evaluate(model))
        line = f'lr {lr:g}: final loss {final_loss(curves[lr]):.3f}'
        if scores[lr]:
            line += f', {metric} {scores[lr][-1]:.3f}'
        if verbose:
            print(line)
    return curves, scores

lm_seed = random.getrandbits(31)
sgd_lm, _ = lr_sweep(lambda: TinyLM(len(data.vocab)),
                     lambda model, lr: torch.optim.SGD(model.parameters(), lr,
                                                       momentum=0.9),
                     data, lrs=[0.03, 0.1, 0.3, 1.0], seed=lm_seed)
```

```
lr 0.03: final loss 2.240
lr 0.1: final loss 1.867
lr 0.3: final loss 2.247
lr 1: final loss inf
```

```
adam_lm, _ = lr_sweep(lambda: TinyLM(len(data.vocab)),
                      lambda model, lr: torch.optim.Adam(model.parameters(),
                                                         lr),
                      data, lrs=[3e-4, 1e-3, 3e-3, 1e-2], seed=lm_seed)
```

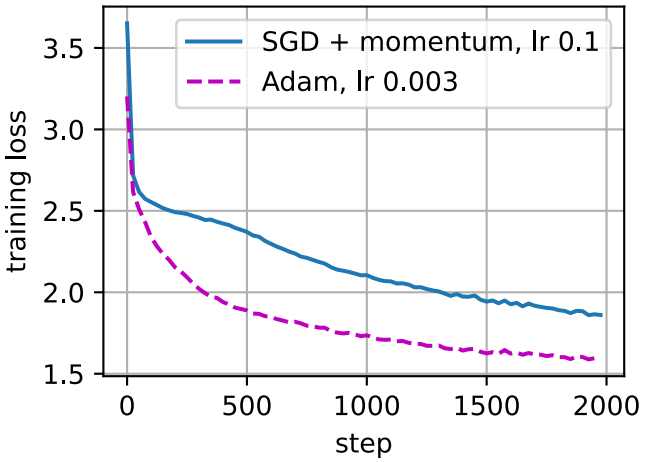
```
lr 0.0003: final loss 1.926
lr 0.001: final loss 1.672
lr 0.003: final loss 1.594
lr 0.01: final loss 1.648
```

SGD's best learning rate sits directly below one at which training diverges, printed as `inf`. Adam's best rate is interior, with stable neighbors on both sides. We plot the two tuned runs:

```
best_sgd = min(sgd_lm, key=lambda lr: final_loss(sgd_lm[lr]))
best_adam = min(adam_lm, key=lambda lr: final_loss(adam_lm[lr]))
d2l.plot(list(range(0, 2000, 25)),
         [smooth(sgd_lm[best_sgd]), smooth(adam_lm[best_adam])],
         'step', 'training loss',
```

```
          legend=[f'SGD + momentum, lr {best_sgd:g}',
                  f'Adam, lr {best_adam:g}'])
loss_sgd = final_loss(sgd_lm[best_sgd])
loss_adam = final_loss(adam_lm[best_adam])
print(f'final loss: SGD {loss_sgd:.3f}, Adam {loss_adam:.3f}, '
      f'gap {loss_sgd - loss_adam:.3f} nats')
print(f"Adam reaches SGD's final loss at step "
      f'{steps_to_reach(adam_lm[best_adam], loss_sgd)} of 2000')
```

```
final loss: SGD 1.867, Adam 1.594, gap 0.273 nats
Adam reaches SGD's final loss at step 600 of 2000
```



Tuned Adam ends about 0.03 to 0.2 nats below tuned SGD, a perplexity 3 to 20% lower, and its curve lies below SGD's from the first hundred steps to the end. No learning rate in SGD's grid closes the gap: the best one is the last stable one. Repeated with fresh random draws, both tuned losses move by a few hundredths of a nat, and SGD, run just below the rate at which it diverges, can end up to about a tenth of a nat higher. The last line of the output measures the gap in time rather than in loss. Adam reaches the loss on which SGD ends 40% to 80% of the way through the run, so it optimizes this model about 1.3 to 2.5 times as fast.

9.5.3.2 Comparison on a CNN

The same protocol trains `FashionCNN`. Adam's grid stays the same, SGD's grid moves one step lower, and every run also records its test accuracy.

```
def run_cnn(make_optimizer, lrs, seed):
    """lr_sweep of FashionCNN, with the test accuracy at the end."""
    return lr_sweep(FashionCNN, make_optimizer, fashion, lrs, seed,
                    evaluate=lambda model: evaluate_accuracy(
                        model, fashion.val_dataloader()),
                    metric='test accuracy')

seed = random.getrandbits(31)
```

```
sgd_cnn, sgd_cnn_acc = run_cnn(
    lambda model, lr: torch.optim.SGD(model.parameters(), lr, momentum=0.9),
    lrs=[0.01, 0.03, 0.1, 0.3], seed=seed)
```

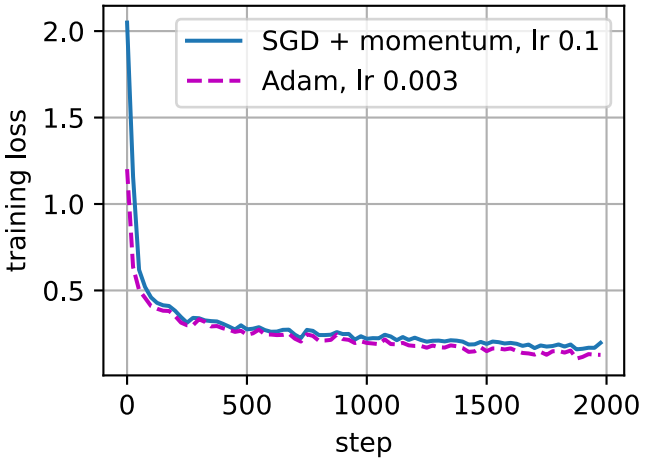
```
lr 0.01: final loss 0.293, test accuracy 0.893
lr 0.03: final loss 0.217, test accuracy 0.901
lr 0.1: final loss 0.174, test accuracy 0.901
lr 0.3: final loss 2.305, test accuracy 0.100
```

```
adam_cnn, adam_cnn_acc = run_cnn(
    lambda model, lr: torch.optim.Adam(model.parameters(), lr),
    lrs=[3e-4, 1e-3, 3e-3, 1e-2], seed=seed)
```

```
lr 0.0003: final loss 0.294, test accuracy 0.881
lr 0.001: final loss 0.202, test accuracy 0.906
lr 0.003: final loss 0.126, test accuracy 0.918
lr 0.01: final loss 0.164, test accuracy 0.898
```

```
best_sgd_cnn = min(sgd_cnn, key=lambda lr: final_loss(sgd_cnn[lr]))
best_adam_cnn = min(adam_cnn, key=lambda lr: final_loss(adam_cnn[lr]))
d2l.plot(list(range(0, 2000, 25)),
         [smooth(sgd_cnn[best_sgd_cnn]), smooth(adam_cnn[best_adam_cnn])],
         'step', 'training loss',
         legend=[f'SGD + momentum, lr {best_sgd_cnn:g}',
                 f'Adam, lr {best_adam_cnn:g}'])
loss_sgd = final_loss(sgd_cnn[best_sgd_cnn])
loss_adam = final_loss(adam_cnn[best_adam_cnn])
print(f'final loss: SGD {loss_sgd:.3f}, Adam {loss_adam:.3f}, '
      f'gap {loss_sgd - loss_adam:.3f} nats')
print(f"Adam reaches SGD's final loss at step "
      f'{steps_to_reach(adam_cnn[best_adam_cnn], loss_sgd)} of 2000')
print(f'test accuracy: SGD {sgd_cnn_acc[best_sgd_cnn][-1]:.3f}, '
      f'Adam {adam_cnn_acc[best_adam_cnn][-1]:.3f}')
```

```
final loss: SGD 0.174, Adam 0.126, gap 0.048 nats
Adam reaches SGD's final loss at step 1275 of 2000
test accuracy: SGD 0.901, Adam 0.918
```



Two things repeat from the language model. SGD's best rate again sits close to the edge of stability: at $\eta = 0.3$ training is unreliable. Often the loss stays at $\log 10 \approx 2.30$, the loss of a uniform guess over the ten classes, and when the run does train, it ends well above the run at $\eta = 0.1$. On both models SGD trains best at or just below the largest rate that trains reliably. And tuned Adam again reaches a lower training loss.

Both losses are cross-entropies in nats per prediction, so the two gaps can be compared directly: a few hundredths to about a tenth of a nat on the CNN, about 0.03 to 0.2 on the language model. A gap of Δ nats means that the better model assigns, on average, e^{Δ} times as much probability to each correct answer: a few percent to about 10% more on the CNN, about 3 to 20% more on the language model.

Measured in time, the question our protocol asks, the two models are close as well: on both, Adam reaches SGD's final loss about 1.3 to 2.5 times as fast. After the first hundred steps the two CNN curves run within about a tenth of a nat of each other, and the test accuracies of the two tuned runs lie within a point or two of each other, between 90% and 93%. On this problem SGD with momentum is a competitive choice, which is why it served computer vision for a decade.

9.5.3.3 Sources of the Gap

On our two small models Adam's speedup is modest and of similar size. On large transformers it is decisive: SGD with momentum falls far behind Adam there, while on CNNs of comparable scale the two stay close (Y. Zhang et al. 2024). The reasons are still being researched, but three findings point in the same direction.

First, the gap is not about minibatch noise. Kunstner et al. (2023) found that it persists in *full-batch* training, where there is no noise to average away, and that Adam's advantage follows that of *sign descent*, the method that moves each coordinate by a fixed amount in the direction opposite to the sign of its gradient. The connection is visible in Equation 9.29: with both averages switched off ($\beta_1 = \beta_2 = 0$) and $\epsilon = 0$, the update becomes $\eta\, \mathbf{g}_t / |\mathbf{g}_t| = \eta\, \mathrm{sign}(\mathbf{g}_t)$. Adam is a smoothed sign method. On these problems it mostly equalizes the step across coordinates whose gradients differ by orders of magnitude.

Second, the differences being equalized come from the data. Word frequencies are heavy-tailed: most words are rare, so the embedding rows and output weights that serve them see

small, infrequent gradients, while those of frequent words see large, constant ones. Kunstner et al. (2024) showed that gradient descent makes almost no progress on the loss of the rare classes while Adam makes progress on all of them, even in linear models with imbalanced classes.

Third, they come from the architecture. Across the blocks of a transformer, the embedding, attention, and MLP parameters have curvatures so different that no single learning rate suits them all, whereas the blocks of a CNN look much more alike. A single learning rate must be small enough for the stiffest block, which slows all the others. A per-coordinate method gives every block its own effective step. The three parameter groups of `TinyLM` are a coarse version of this division: groups with different roles and different gradient statistics, all sharing one η under SGD.

How much each finding explains for `TinyLM` is open, and the model is a mild case of the last two. Its bytes are skewed, but the rare ones carry little of the loss. Every lowercase letter appears more than twice in an average minibatch of 4,096 predictions. About a hundred byte values, mostly pieces of the UTF-8 encodings of non-English characters, appear less than once in ten minibatches, but together they make up less than 0.1% of the text, whereas the rare words of a word-level vocabulary together make up a sizable share of it. And with two blocks of width 128 the model is far smaller than the transformers of these studies. The exercises test both explanations on this model.

9.5.3.4 Limits of the Variance Estimate

Adam's denominator $\sqrt{\hat{\mathbf{v}}_t}$ is an estimate built from one noisy stream of gradients, and its errors are not symmetric: an estimate that is too small produces a step that is too large. The worst case is a coordinate whose gradient is usually small and occasionally large. Between the large gradients the average $\mathbf{v}_t$ decays toward the small ones, so when the next large gradient arrives, its step is far larger than the scale of that coordinate warrants. Reddi et al. (2018) built a convex online problem from this pattern on which Adam converges to the *worst* feasible point. The problem is adversarial and says nothing about ordinary training data, but it shows that Adam has no general convergence guarantee. Their repair, AMSGrad, divides by the running maximum of $\hat{\mathbf{v}}_t$, so that the estimate can never shrink. Section 35.5.2.2 works through the construction and the repair. Neither AMSGrad nor later repairs replaced Adam in practice.

The constant ϵ limits the damage. It is meant to guard against division by zero, but it also acts as a floor on the denominator of Equation 9.29. The learning rate of a coordinate, the factor $\eta/(\sqrt{\hat{v}_{t,i}} + \epsilon)$ that multiplies its momentum $\hat{m}_{t,i}$, therefore never exceeds η/ϵ. A small ϵ trusts the variance estimate even when it is tiny; a large ϵ does not. In the limit of large ϵ the denominator is ϵ for every coordinate, and Adam becomes $\mathbf{x}_{t+1} = \mathbf{x}_t - (\eta/\epsilon)\,\hat{\mathbf{m}}_t$. Since $\mathbf{m}_t$ is $1 - \beta_1$ times the momentum buffer of Section 9.4, this is SGD with momentum $\beta = \beta_1$ and learning rate $(1 - \beta_1)\eta/\epsilon$, up to the bias correction of the first steps. Values of ϵ between the default 10^{-8} and this limit interpolate between the two methods, which makes ϵ a tuning knob rather than a numerical detail.

9.5.4 Decoupled Weight Decay

Adam is almost always trained with weight decay, which shrinks the weights toward zero at every step (Section 2.7). Under SGD it makes no difference whether the shrinkage is written as a penalty $\frac{\lambda}{2}\|\mathbf{x}\|^2$ added to the loss or applied to the weights directly. Under Adam it

does: the gradient of the penalty passes through the per-coordinate scaling, and the same λ then shrinks every coordinate by a different amount. AdamW (Loshchilov and Hutter 2019) applies the decay outside the scaling and restores one meaning for λ.

9.5.4.1 Penalty and Preconditioner

Adding the penalty to the loss adds $\lambda \mathbf{x}_t$ to the gradient. Under SGD the update then splits into a uniform shrinkage and an ordinary gradient step:

$$\mathbf{x}_{t+1} = \mathbf{x}_t - \eta\,(\mathbf{g}_t + \lambda \mathbf{x}_t) = (1 - \eta\lambda)\,\mathbf{x}_t - \eta\,\mathbf{g}_t. \tag{9.30}$$

Penalizing the loss *is* shrinking the weights, by the factor $1 - \eta\lambda$ for every coordinate. Now feed the same penalized gradient to Adam. Both moments are built from it, so the penalty passes through the preconditioner too. If we treat $\sqrt{\hat{\mathbf{v}}_t}$ as fixed, the shrinkage of coordinate i in one step compares with that of weight decay as follows:

$$\underbrace{\frac{\eta\,\lambda}{\sqrt{\hat{v}_{t,i}} + \epsilon}\,x_{t,i}}_{\ell_2 \text{ penalty through Adam}} \qquad \text{versus} \qquad \underbrace{\eta\,\lambda\,x_{t,i}}_{\text{weight decay}}\ . \tag{9.31}$$

The penalty is rescaled per coordinate by the same factor that rescales the loss gradient: a parameter with large gradients (large $\hat{v}_i$) is barely decayed, while a parameter with small gradients is decayed strongly. Whatever λ we choose, Adam changes its effective value by coordinate and over time. Section 35.5.3 follows the exact recurrence and shows two coordinates with the same λ whose effective decay rates differ by a factor of 100.

AdamW *decouples* the two: the loss gradient goes through the preconditioner, and the decay does not,

$$\mathbf{x}_{t+1} = (1 - \eta\lambda)\,\mathbf{x}_t - \eta\,\frac{\hat{\mathbf{m}}_t}{\sqrt{\hat{\mathbf{v}}_t} + \epsilon}, \tag{9.32}$$

with $\hat{\mathbf{m}}_t$ and $\hat{\mathbf{v}}_t$ computed from the plain gradient $\mathbf{g}_t$ as in Equation 9.28. The shrinkage is the uniform factor of Equation 9.30 again. When the learning rate follows a schedule η_t, the decay should follow it, so that the weights stop shrinking when learning stops. Loshchilov and Hutter (2019) multiply the decay by the schedule's multiplier; the library implementations we use multiply it by the whole learning rate, as in Equation 9.32, and we follow them.

9.5.4.2 AdamW from Scratch

The decoupled update adds one operation to Adam: shrink the weights, then take the Adam step.

```
class AdamW(Adam):
    """Adam with decoupled weight decay: shrink the weights, then step."""
    def __init__(self, params, lr=1e-3, weight_decay=0.01):
        super().__init__(params, lr)
        self.defaults['weight_decay'] = weight_decay  # For later groups
```

```
        for group in self.param_groups:
            group.setdefault('weight_decay', weight_decay)

    @torch.no_grad()
    def step(self):
        for group in self.param_groups:
            for p in group['params']:
                if p.grad is not None:
                    p.mul_(1 - group['lr'] * group['weight_decay'])
        super().step()
```

`torch.optim.Adam` also accepts a `weight_decay` argument. It adds $\lambda \mathbf{x}$ to the gradient before the moments are updated, which is the coupled form on the left of Equation 9.31. `torch.optim.AdamW` is the decoupled form.

We check it against the library's AdamW as before, at $\lambda = 0.1$:

```
compare(partial(AdamW, weight_decay=0.1),
        partial(torch.optim.AdamW, weight_decay=0.1), lr=0.01)
```

```
first step / lr: 1.0037 to 1.0198
largest difference after 100 steps: 5.6e-09
```

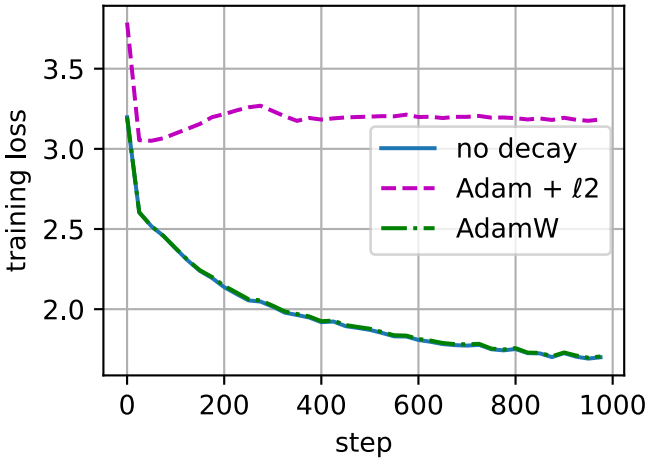
The first step now also shrinks every weight by the factor $1 - \eta\lambda$, so it is no longer exactly η, and the two implementations again agree to within rounding.

9.5.4.3 Coupled and Decoupled Decay Compared

How large is the difference on a real model? We train `TinyLM` for 1,000 steps at the learning rate $\eta = 0.003$ and the common choice $\lambda = 0.1$, once with the penalty inside Adam's gradient and once decoupled, and also once without any decay for reference. All three runs start from the same random initialization.

```
seed = random.getrandbits(31)
curves = {}
for name, make_optimizer in [
        ('no decay',
         lambda model, lr: torch.optim.Adam(model.parameters(), lr)),
        ('Adam + ℓ2', lambda model, lr: torch.optim.Adam(
            model.parameters(), lr, weight_decay=0.1)),
        ('AdamW', lambda model, lr: torch.optim.AdamW(
            model.parameters(), lr, weight_decay=0.1))]:
    print(f'{name}: ', end='')
    curves[name] = lr_sweep(lambda: TinyLM(len(data.vocab)), make_optimizer,
                            data, [0.003], seed, num_steps=1000)[0][0.003]
d2l.plot(list(range(0, 1000, 25)), [smooth(c) for c in curves.values()],
         'step', 'training loss', legend=list(curves))
```

```
no decay: lr 0.003: final loss 1.705
Adam + ℓ2: lr 0.003: final loss 3.183
AdamW: lr 0.003: final loss 1.711
```



The decoupled run trains almost as if there were no decay. The coupled run ends more than a nat higher, and after an early dip its loss rises again. To read its level, we compare it with a model that ignores the context and predicts every byte from its frequency in the training text. The loss of that model is the entropy of the byte frequencies:

```
freq = np.bincount(data.train, minlength=256) / len(data.train)
freq = freq[freq > 0]
print(f'entropy of the byte frequencies: {-(freq * np.log(freq)).sum():.2f}')
```

```
entropy of the byte frequencies: 3.17
```

The coupled run ends close to this value: it has lost what it learned about the context in its first hundred steps. Equation 9.31 explains the rise. As training makes progress the loss gradients shrink, $\sqrt{\hat{\mathbf{v}}}$ shrinks with them, and the penalty, divided by that small number, grows by orders of magnitude. Once $\lambda\mathbf{x}$ dominates the gradient, Adam normalizes it like any other gradient, so every coordinate shrinks by about η per step whatever the value of λ: past this point, changing λ hardly changes the run.

The coupled form needs a much smaller λ, which is easy to find. The harder question is how the best λ moves when the learning rate changes: if it moves, the two settings must be searched together on a grid instead of one after the other. We measure the held-out loss on a grid of three learning rates and several values of λ for each form, from 10^{-4} to 10^{-2} for the coupled form and from 0.3 to 10 for the decoupled one. In a run this short, decay helps only where there is overfitting to prevent, and on WikiText-103 there is none: 800 steps read about 1% of the training text. For this grid we therefore make the model overfit on purpose, on the kind of text that it would memorize: the first 8,000 windows of *The Time Machine*, split into characters as in Section 8.2. In 800 steps the model sees each window about six times, and we measure the held-out loss on the 2,000 windows that follow.

```
small = d2l.TimeMachine(batch_size=64, num_steps=64, tokenization='char',
                        num_train=8000, num_val=2000)
```

The function `sweep` trains one model per pair (η, λ), with a quiet `lr_sweep` for each λ, and returns the table of held-out losses, one row per learning rate. The held-out loss is the mean

cross-entropy over the validation windows, the logarithm of `lm_perplexity`.

```
lrs = [1e-3, 3e-3, 1e-2]
wds_coupled, wds_decoupled = [1e-4, 3e-4, 1e-3, 3e-3, 1e-2], [0.3, 1, 3, 10]

def sweep(make_optimizer, data, lrs, wds, seed, num_steps=800):
    """Held-out loss for each pair (lr, wd), one row per learning rate."""
    held_out = lambda model: math.log(lm_perplexity(model,
                                                    data.val_dataloader()))
    columns = [lr_sweep(lambda: TinyLM(len(data.vocab)),
                        lambda model, lr: make_optimizer(model, lr, wd),
                        data, lrs, seed, num_steps, evaluate=held_out,
                        verbose=False)[1]
               for wd in wds]
    table = [[column[lr][-1] for column in columns] for lr in lrs]
    for lr, row in zip(lrs, table):
        print(f'lr {lr:g}: ' + ' '.join(f'{v:.2f}' for v in row))
    return table

seed = random.getrandbits(31)
coupled_tbl = sweep(lambda model, lr, wd: torch.optim.Adam(
    model.parameters(), lr, weight_decay=wd), small, lrs, wds_coupled, seed)
```

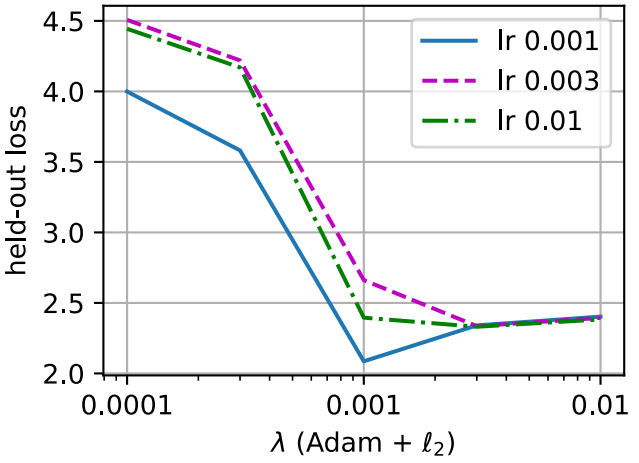
```
lr 0.001: 4.00 3.58 2.09 2.34 2.40
lr 0.003: 4.51 4.22 2.66 2.34 2.39
lr 0.01: 4.44 4.17 2.40 2.33 2.38
```

```
decoupled_tbl = sweep(lambda model, lr, wd: torch.optim.AdamW(
    model.parameters(), lr, weight_decay=wd), small, lrs, wds_decoupled,
    seed)
```

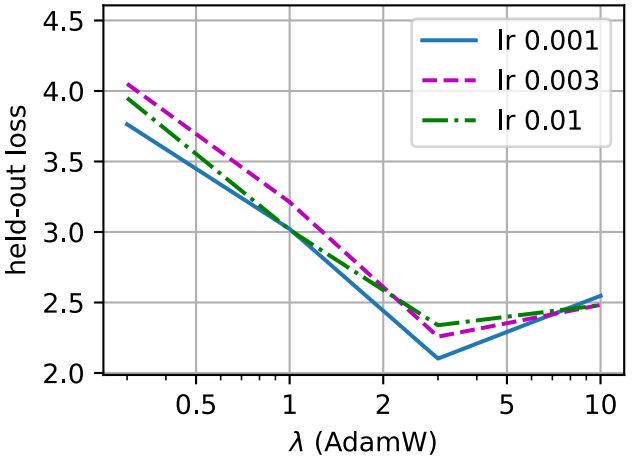
```
lr 0.001: 3.76 3.02 2.10 2.55
lr 0.003: 4.05 3.21 2.26 2.48
lr 0.01: 3.95 3.02 2.34 2.48
```

We plot the held-out loss against λ, one curve per learning rate, for the coupled form and then for the decoupled form, on a common vertical scale.

```
values = [v for table in (coupled_tbl, decoupled_tbl)
          for row in table for v in row]
ylim = (min(values) - 0.1, max(values) + 0.1)
legend = [f'lr {lr:g}' for lr in lrs]
d2l.plot(wds_coupled, coupled_tbl, '$\\lambda$ (Adam + $\\ell_2$)',
         'held-out loss', legend=legend, ylim=ylim, xscale='log')
```




```
d2l.plot(wds_decoupled, decoupled_tbl, '$\\lambda$ (AdamW)',
         'held-out loss', legend=legend, ylim=ylim, xscale='log')
```



In this grid the AdamW curves have their minimum at about the same λ, 3 or 10, for all three learning rates: changing η moves a curve up or down but hardly sideways. The coupled form works with values of λ about three orders of magnitude smaller. At its largest λ the coupled curves end at nearly the same loss for every learning rate, the saturation described above. Below that, the best λ differs among the three learning rates. Tuned fully, the two forms reach similar held-out losses. They differ in the shape of the search: a column for AdamW, a whole grid for the coupled form.

A fixed point explains why the best λ of AdamW stays put here. Where the weights stop moving on average, the shrinkage in Equation 9.32 balances the Adam step, $\eta\lambda\,\mathbf{x} = -\eta\,\hat{\mathbf{m}}/(\sqrt{\hat{\mathbf{v}}}+\epsilon)$. The learning rate cancels: λ alone selects the solution near which the run settles, and η sets how fast the run gets there, through the factor $1-\eta\lambda$ per step. Near the best λ, every run of the grid lasts longer than $1/(\eta\lambda)$ steps, so every run gets close. The argument depends on the parameterization, in which both terms scale with η, and on the

regime, as the next subsection shows.

9.5.4.4 Weight Decay in Large-Scale Training

Our grid used weight decay against overfitting, the role it played in Section 2.7, where it acts as a prior that pulls the model toward small weights. Large language models are trained for about one pass over data that they never see twice, so there is little overfitting to fight, yet most published recipes still use $\lambda = 0.1$. The decay does a different job there (D'Angelo et al. 2024; Kosson et al. 2024).

The mechanism runs through the norms of the weights. A weight matrix whose output passes through a normalization layer is *scale-invariant*: multiplying it by a constant does not change the network's function, because the normalization divides the constant out. Only the direction of such a matrix matters, and an update of fixed size turns it by an angle roughly proportional to the update divided by the matrix's norm. In a pre-norm transformer this holds only approximately, since the residual stream carries each block's output around the next normalization, but it describes enough of the parameters to set the dynamics. Gradient noise makes the norms grow, decay makes them shrink, and the two settle into an equilibrium in which each weight turns by a roughly constant angle per step (Kosson et al. 2024). Through this equilibrium the product $\eta\lambda$ sets the *effective* learning rate, the angle per step, and sets it alike for all layers. This is a large part of why decay in large language-model runs lowers the training loss instead of trading training loss for held-out loss (D'Angelo et al. 2024). Small weights also keep low-precision training in a safe numerical range.

With decay multiplied by the learning rate, each step multiplies the weights by $1 - \eta\lambda$, so a contribution made k steps ago has shrunk by the factor $(1 - \eta\lambda)^k \approx e^{-\eta\lambda k}$. An AdamW iterate is therefore approximately an average of its recent updates over a horizon of $1/(\eta\lambda)$ steps: with $\eta = 0.003$ and $\lambda = 0.1$, about 3,300 steps. Measured in passes over a dataset of D examples with batch size B, the horizon is $\tau = B/(\eta\lambda D)$. Bergsma et al. (2025a) found that the best τ stays roughly constant as models grow at a fixed ratio of training tokens to parameters, and follows a power law in that ratio otherwise, so that λ can be set from B, D, and the model size rather than tuned again. The horizon also explains why η and λ act on the equilibrium through their product.

Keeping the product fixed when η changes is the opposite of what our grid showed. The two results describe different regimes. A scale-invariant matrix has no fixed point of the kind found in the previous subsection: its gradient is orthogonal to it, so decay and updates balance only in the norm, which the network ignores. What remains is the angle per step, set by $\eta\lambda$. When most of the decayed weights are of this kind, the quantity to keep when η is retuned is the product, which is also how Loshchilov and Hutter (2019) parameterize the decay: they multiply it by the schedule only, not by η, and on image classifiers with batch normalization they found its best value nearly independent of the learning rate. When decay mainly fights overfitting, as in our grid, λ itself selects the solution. The exercise on the fixed-product ridge measures which of the two holds for `TinyLM`.

9.5.4.5 Parameters Exempt from Decay

The scale-invariance argument covers only matrices whose output is normalized, and standard practice decays only the matrices among the three parameter groups of `TinyLM`. The other two groups are exempt for different reasons. Biases and LayerNorm scales and shifts are few, and they set the scale and offset of each feature directly, so shrinking them toward zero distorts the function instead of controlling the dynamics. Biases were already

exempt in Section 2.7. Embedding rows are sparse: a row receives a gradient only when its token occurs, but the decay acts at every step, so the rows of rare tokens are decayed without any signal to balance it. OLMo 2 traced a training instability to this: decay shrank the embeddings until the LayerNorm after them, which divides by their norm, amplified their gradients (Team OLMo et al. 2025).

The implementation uses two parameter groups, one decayed and one not, and the split is the one we counted for `TinyLM`.

```
torch.manual_seed(lm_seed)  # The initialization of the comparison's runs
model = TinyLM(len(data.vocab))
decay = [p for name, p in model.named_parameters()
         if p.ndim == 2 and 'emb' not in name]
no_decay = [p for name, p in model.named_parameters()
            if p.ndim != 2 or 'emb' in name]
optimizer = torch.optim.AdamW([
    {'params': decay, 'weight_decay': 0.1},
    {'params': no_decay, 'weight_decay': 0.0}], lr=best_adam)
n_decay = sum(p.numel() for p in decay)
n = n_decay + sum(p.numel() for p in no_decay)
print(f'decayed: {n_decay} of {n} parameters')
```

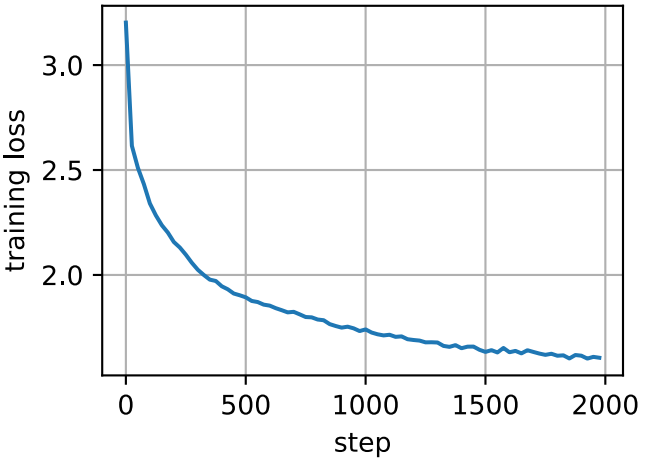
```
decayed: 425984 of 470784 parameters
```

`torch.optim.AdamW` takes a list of parameter groups, each a dictionary with its own settings; settings not given in a group fall back to the keyword arguments.

About 90% of the parameters are decayed. We train this configuration for 2,000 steps at Adam's tuned learning rate, from the same initialization as the runs of the language-model comparison, and compare it with Adam's tuned run there, which used no decay:

```
losses = train_steps(model, data, optimizer, num_steps=2000)
d2l.plot(list(range(0, 2000, 25)), [smooth(losses)],
         'step', 'training loss')
print(f'perplexity: AdamW {math.exp(final_loss(losses)):.2f}, '
      f'Adam without decay {math.exp(final_loss(adam_lm[best_adam])):.2f}')
```

```
perplexity: AdamW 4.99, Adam without decay 4.93
```



AdamW ends within a few percent of the run without decay, in either direction. In 2,000 steps the model reads about 3% of the training text and does not overfit, so the decay, whose horizon $1/(\eta\lambda)$ is longer than the run, has nothing to prevent and cannot help. This is the configuration in which Adam is used today: AdamW, decay on the matrices only, and λ chosen together with the schedule. What remains is its cost in memory.

9.5.5 Optimizer-State Memory

Adam and AdamW store two numbers per parameter, m and v, on top of the weights and the gradients. At the size of `TinyLM` that is invisible; at the size of a language model it decides what fits on a device. We count the bytes per parameter for three common setups and multiply by the parameter count n of `TinyLM`, which `d2l.num_params` returns.

In full fp32 training, a parameter costs 4 bytes each for the weight, the gradient, m, and v: 16 bytes, half of them for m and v. Mixed precision does not reduce this. The forward and backward passes use bf16 weights and gradients, 2 bytes each, but stable training keeps an fp32 *master copy* of the weights for the optimizer to update, so the total is again 16 bytes. Many setups also keep an fp32 copy of the gradients to accumulate them over several mini-batches, which brings the total to 20 (Section 39.6).

```
n = d2l.num_params(model)  # The TinyLM just trained
setups = {'fp32 weights, grads, m, v': (4, 4, 8),
          'bf16 weights + grads, fp32 master, m, v': (2, 2, 12),
          'the same + fp32 gradient accumulator': (2, 6, 12)}
print(f'{"":<42}{"B/param":>8}{"TinyLM":>10}{"7B model":>10}')
for name, (w, g, s) in setups.items():
    per = w + g + s
    print(f'{name:<42}{per:>8}{n * per / 1e6:>8.1f}MB'
          f'{7e9 * per / 1e9:>8.0f}GB')
```

```
                                          B/param    TinyLM  7B model
fp32 weights, grads, m, v                      16     7.5MB     112GB
bf16 weights + grads, fp32 master, m, v        16     7.5MB     112GB
the same + fp32 gradient accumulator           20     9.4MB     140GB
```

For `TinyLM` the total is a few megabytes. The same arithmetic for a model with 7 billion parameters gives 112 to 140 GB before a single activation is stored, more than an 80-GB accelerator holds. Of the 20 bytes, 12 are the fp32 master copy, m, and v, which only the optimizer reads. This is why optimizer state is the first target when memory runs out. It can be stored in fewer bytes or split across the devices of a data-parallel group instead of copied to each, the systems techniques of Chapter 13.

9.5.6 Summary

A single learning rate must suit the stiffest direction of the loss. Adam gives each coordinate its own step, scaled by an estimate of that coordinate's gradient size. The estimate is AdaGrad's running sum of squared gradients, made to forget by RMSProp's exponential average. Adam combines it with momentum and corrects both averages exactly for their start at zero. On a small transformer language model and on a CNN of the same size, tuned Adam reaches tuned SGD's final training loss about 1.3 to 2.5 times as fast and ends a few hundredths to about two tenths of a nat lower; on the CNN the test accuracies agree to within a point or two. On large transformers the gap is far wider, and the best current explanations point to gradient scales that differ across coordinates, from heavy-tailed data and from dissimilar blocks of the architecture. Under Adam an ℓ_2 penalty is not weight decay, because its gradient passes through the per-coordinate scaling. AdamW applies the decay outside it, which gives λ the same meaning for every coordinate. At scale the product $\eta\lambda$ sets the effective learning rate of normalized layers. Adam stores two extra numbers per parameter, which in mixed precision bring a parameter to 16 to 20 bytes. What Adam does not remove is the choice of η over time, the subject of Section 9.6.

9.5.7 Exercises

1. code **Rotating the coordinates.** Per-coordinate methods are axis-aligned. Rotating the valley `f_valley` by 45° gives

$$f(\mathbf{x}) = 0.05(x_1 + x_2)^2 + (x_1 - x_2)^2,$$

   whose Hessian has the same eigenvalues as that of `f_valley`, 0.2 and 4.

   1. Write the gradient of the rotated valley as a function like `valley_grad`, and pass it as `f_grad` to `adagrad_2d`, `rmsprop_2d`, and the `gd_2d` of Section 9.2. On both valleys, run AdaGrad at $\eta = 0.4$ and $\eta = 2$, RMSProp at $\eta = 0.4$, and gradient descent at $\eta = 0.4$ with `d2l.train_2d` and `steps=2000`. For each run, record the first step at which $\|\mathbf{x}\| \le 10^{-3}$, or that no step within the 2,000 reaches it.
   2. How much of the adaptive methods' advantage over gradient descent survives the rotation? Explain the answer from the Hessian of the rotated valley.

2. **Bias correction by initialization.** Instead of the zero initialization and the corrections in Equation 9.28, initialize $\mathbf{m}_1 = \mathbf{g}_1$ and $\mathbf{v}_1 = \mathbf{g}_1^2$ and use the uncorrected averages from then on.

   1. Show that the two schemes coincide at $t = 1$ and derive the weights that each assigns to $\mathbf{g}_1, \dots, \mathbf{g}_t$ for $t > 1$.
   2. What is lost relative to the exact correction, and for which range of t does the difference matter?

3. [extended] **Frequent and rare tokens.** Following Kunstner et al. (2024), split the 256 byte values of `data` into a frequent and a rare group by their share of the training text,

for instance at 0.1%. Under SGD with momentum and under Adam, each at the best learning rate of `sgd_lm` and `adam_lm`, record every 100 steps the loss of `TinyLM` on the targets of each group, measured on a fixed set of validation windows.

1. Which optimizer makes progress on the rare group, and does the gap of Section 9.5.3 come mainly from the rare group, from the frequent group, or from both?
2. Repeat the measurement with subword tokens, among which rare tokens make up a larger share of the text. Train a `d2l.BPETokenizer` with 1,024 tokens, the GPT-2 pattern, and no special tokens (Section 8.2) on the first million bytes of the training text, encode the first ten million bytes and the validation split with it, store the results in `data.train` and `data.val`, and train `TinyLM` with a vocabulary of 1,024. Compare the two splits.

4. code **Per-group learning rates for SGD.** ⊠ The three parameter groups of `TinyLM` in Section 9.5.2, embeddings, matrices, and vectors, share one learning rate under SGD with momentum.

   1. Give each group its own learning rate. Starting from the best rate of `sgd_lm`, scale one group's rate by each factor in $\{1/10, 1/3, 3, 10\}$ while holding the other two fixed, and keep the best factor for each group. Then train once with the three best factors combined.
   2. How much of the gap between the best runs of `sgd_lm` and `adam_lm` does per-group SGD close? What does the result say about whether Adam's advantage on this model comes from scale differences between the three groups or from differences among the coordinates within each group?

5. code **Two repairs of the variance estimate.** Adam's second-moment update can be written as

$$\mathbf{v}_t = \mathbf{v}_{t-1} + (1-\beta_2)(\mathbf{g}_t^2 - \mathbf{v}_{t-1}), \tag{9.33}$$

which shrinks $\mathbf{v}_t$ whenever $\mathbf{g}_t^2 < \mathbf{v}_{t-1}$. Yogi (Zaheer et al. 2018) limits how fast it can shrink by replacing the increment with

$$(1-\beta_2)\,\mathbf{g}_t^2 \odot \operatorname{sgn}(\mathbf{g}_t^2 - \mathbf{v}_{t-1}). \tag{9.34}$$

AMSGrad (Reddi et al. 2018) carries the running maximum

$$\hat{\mathbf{v}}_t^{\max} = \max(\hat{\mathbf{v}}_{t-1}^{\max}, \hat{\mathbf{v}}_t) \tag{9.35}$$

and uses it in place of $\hat{\mathbf{v}}_t$ in Equation 9.29. (The function `online` below takes the maximum of the uncorrected $\mathbf{v}_t$ instead, as the original paper does; implement the version written here.)

   1. Implement both by changing the from-scratch Adam of this section, and check each against a hand computation of its first three steps on a scalar gradient sequence such as $(1, 0.1, 0.1)$.
   2. The function `online` of Section 35.5.2.2 runs Adam and AMSGrad on the losses Equation 35.54, on which Adam converges to the worst point $x = +1$. Add the Yogi update to it, run it at $C = 4$, and report where its iterate ends. Explain the outcome from how far the v_t of Adam and of Yogi can fall during the two -1 gradients of each period.

6. code **The ϵ ceiling.** Sweep $\epsilon \in \{10^{-8}, 10^{-6}, 10^{-4}, 10^{-2}, 1\}$ for Adam on `TinyLM` at the best learning rate of `adam_lm`. Explain the trend at both ends of the sweep with

the ceiling η/ϵ on each coordinate's learning rate of Section 9.5.3.4, and identify the ϵ from which the run resembles SGD with momentum, as in `sgd_lm`, at learning rate $(1-\beta_1)\eta/\epsilon$.

7. code **Saturation of coupled decay.** The experiment of Section 35.5.3 starts two parameters at $x_i = 1$ and gives them pure-noise loss gradients $g_i \sim \mathcal{N}(0, \sigma_i^2)$ with $\sigma = (10, 0.1)$. At $\eta = 10^{-3}$ and $\lambda = 0.1$, Adam with an ℓ_2 penalty decays them at the rates $\eta\lambda/\sigma_i$ that Equation 9.31 predicts.
   1. In the opposite limit $\sigma_i = 0$, the penalized gradient is $\lambda x_{t,i}$ alone. Show that for $\epsilon = 0$ the trajectory of $x_{t,i}$ under Adam with an ℓ_2 penalty does not depend on $\lambda > 0$.
   2. Extend the experiment to $\sigma_i \in \{10, 1, 0.1, 0.01, 0.001\}$ at $\eta = 10^{-3}$ and record $|x_i|$ after 1,000 steps of Adam with an ℓ_2 penalty for each $\lambda \in \{0.1, 1, 10\}$. Which coordinates no longer respond to λ, and how does the transition between the two regimes depend on σ_i and λ?

8. code **The fixed-product ridge.** ⊠ Both library implementations used in this section multiply the decay by the full learning rate, so the per-step shrinkage is $s = \eta\lambda$. Section 9.5.4.4 argues that at scale η and λ act through this product, which sets an averaging horizon of $1/s$ steps. Loshchilov and Hutter (2019) instead scale the decay by the schedule multiplier only, so that under a constant schedule their decay coefficient is s itself.
   1. Each column of a grid can fix s instead of λ. Use `decoupled_tbl` to predict where the best entry of each row falls on the shrinkage axis `[1e-3, 3e-3, 1e-2, 3e-2]`. Then rerun the decoupled grid of Section 9.5.4.3 with `sweep` on this axis, passing the decay coefficient to the optimizer as `wd / lr`. Does the best column stay put?
   2. How constant is the held-out loss down the column $s = 3 \times 10^{-3}$? Reconcile the answer with the argument of Section 9.5.4.4, and state which quantity, λ or s, the decay axis of a grid should carry when the learning rate is retuned in this setting.

9. code **Rotational equilibrium.** ⊠ Following Kosson et al. (2024), let $\mathbf{x} \in \mathbb{R}^d$ be a scale-invariant weight vector in the sense of Section 9.5.4.4, trained by AdamW with $\beta_1 = 0$:

$$\mathbf{x}_{t+1} = (1 - \eta\lambda)\,\mathbf{x}_t - \eta\,\mathbf{u}_t, \qquad \mathbf{u}_t = \frac{\mathbf{g}_t}{\sqrt{\hat{\mathbf{v}}_t} + \epsilon}. \tag{9.36}$$

   1. Show that the loss gradient of a scale-invariant weight vector is orthogonal to $\mathbf{x}$. Assume from here on that $\mathbf{u}_t$ is orthogonal to $\mathbf{x}_t$ as well and that $\|\mathbf{u}_t\|^2 \approx d$. Derive, to first order in $\eta\lambda$, the equilibrium norm at which decay balances the growth caused by the updates.
   2. Derive the angle through which $\mathbf{x}$ turns per step at equilibrium. How do the equilibrium norm and the angle each depend on η and λ? Relate the result to the claim of Section 9.5.4.4 that η and λ act through their product.
   3. Check both predictions in NumPy with $d = 1{,}000$ over 20,000 steps from a random unit vector, using pure-noise gradients projected orthogonal to $\mathbf{x}_t$, at $(\eta, \lambda) \in \{(10^{-3}, 0.1), (3 \times 10^{-3}, 0.1), (10^{-3}, 0.3)\}$.

10. **Activations and optimizer state.** Extend the accounting of Section 9.5.5 beyond the `setups` table.
    1. For `TinyLM` with batch size B and sequence length T, estimate the bf16 activation memory that must be stored for the backward pass (per block: the two normal-

ization outputs, the attention inputs and outputs, and the MLP hidden layer). At what batch size do activations overtake the optimizer state?
2. Count the floating-point operations of one AdamW update per parameter and compare them with the roughly $6BT$ operations per parameter of a forward and backward pass over one batch. At which batch size would the optimizer step cost one percent of a training step?

Adapted from Stanford CS336, Assignment 1, Problem adamwAccounting.

9.6 Schedules

The modern recipe of Section 7.6 did not keep the learning rate fixed. It raised the rate linearly over the first few epochs and then let it fall along a cosine curve. Almost every large model today is trained with a warmup followed by a decay. This section asks why the learning rate should change during a run, and what shape the change should have.

The two ends of a run need different things. At the end, a constant learning rate keeps stochastic gradient descent from settling. On the valley $f(\mathbf{x}) = 0.1x_1^2 + 2x_2^2$ with unit gradient noise, the iterates at $\eta = 0.1$ keep wandering around the minimum with a standard deviation of 0.5 along the flat direction, however long they run (Section 9.3). Halving the rate halves the squared radius of that cloud, but it also halves the speed at which the iterates approach it. The start of training has the opposite problem. The convolutional network of this section trains at a learning rate of 7.5 if the rate is raised to that value gradually, while a run that starts at 7.5 stays close to chance accuracy.

A learning-rate *schedule* addresses both problems. It is a function that gives the learning rate η_t at each point t of training, and it can be changed without touching the rest of the training loop. An optimizer makes three decisions (Section 9.1): the descent direction, the step size over time, and the treatment of noise. The schedule is the second. We first ask what a decaying rate has to achieve, on the valley, where the answer can be computed: it must carry the iterate far enough to reach the minimum, and it must end small enough to quench the noise. We then train one small network under the classical decay shapes and under *warmup*, which raises the rate gradually at the start. Last comes the warmup–stable–decay schedule used to train large language models. It holds the rate constant for most of the run, which lets us decide late in training how long to train.

```
import copy
```

9.6.1 Decay on the Valley

At the end of a run a schedule has two jobs: carry the iterate to the minimum and shrink the noise around it. On the valley both effects can be computed in closed form, so we start there.

9.6.1.1 Progress and Noise

On the valley each coordinate of SGD evolves on its own. Take one coordinate with curvature λ and gradient noise ξ_t of mean zero and variance σ^2, and let the learning rate η_t change from step to step. One step of SGD reads

$$x_{t+1} = (1 - \eta_t \lambda)\, x_t - \eta_t\, \xi_t. \tag{9.37}$$

The first term pulls the iterate toward the minimum at 0. The second pushes it in a random direction. Their effects separate. The mean of x_t shrinks by the factor $1 - \eta_s \lambda \approx e^{-\eta_s \lambda}$ at every step s, so after t steps it has shrunk by about $e^{-\lambda B_t}$, where

$$B_t = \sum_{s<t} \eta_s \tag{9.38}$$

is the sum of all learning rates so far, which we call the *budget* of the schedule. For a constant rate, Section 9.3 showed that the noise keeps the iterate in a *noise ball* of squared radius $\eta\sigma^2/(2\lambda)$ around the minimum, for $\eta\lambda \ll 1$. When the rate changes slowly, compared with the $1/(\eta_t\lambda)$ steps the iterate needs to forget its past, the ball follows the current rate. Together,

$$\mathbb{E}\big[x_t^2\big] \approx e^{-2\lambda B_t}\, x_0^2 + \frac{\eta_t\, \sigma^2}{2\lambda}. \tag{9.39}$$

The first term is what remains of the starting distance, and it shrinks with the budget; the second is the noise ball of the current rate. On the flat direction of the valley, $\lambda = 0.2$ and $x_0 = -5$, a budget of $B = 10$ leaves $e^{-2} \approx 14\%$ of the starting distance and a budget of 25 leaves $e^{-5} \approx 0.7\%$. A schedule that converges must therefore do two things: let the budget grow without bound, so that the first term vanishes, and let the rate go to zero, so that the second does.

The conditions of Robbins and Monro (1951) that Section 9.3 stated, $\sum_t \eta_t = \infty$ and $\sum_t \eta_t^2 < \infty$, are a sufficient form of these two requirements. The second is stronger than $\eta_t \to 0$, and it is what their proof for general objectives needs. On the quadratic, Equation 9.39 shows that a rate that goes to zero with an infinite budget already suffices, as long as it falls slowly enough for the noise ball to follow it. Polynomial decay with $\alpha = 1/2$ below confirms this.

Three families of schedules have been used since the early days of stochastic approximation:

$$\begin{aligned} &\eta_t = \eta_i \quad \text{for } t_i \le t < t_{i+1} && \text{piecewise constant,} \\ &\eta_t = \eta_0\, \gamma^t && \text{exponential,} \\ &\eta_t = \eta_0\, (1 + t/\tau)^{-\alpha} && \text{polynomial,} \end{aligned} \tag{9.40}$$

with a decay factor $\gamma \in (0, 1)$, a time scale τ, and an exponent $\alpha > 0$. Piecewise constant decay keeps the rate fixed and cuts it at chosen steps $t_1 < t_2 < \dots$. Exponential decay multiplies the rate by γ at every step, so its budget is the geometric sum $\eta_0/(1-\gamma)$, which stays finite: it violates the first condition. Polynomial decay has an infinite budget for $\alpha \le 1$, and its squares have a finite sum for $\alpha > 1/2$; the case $\alpha = 1$ is the rate $\eta_t \propto 1/t$ of Section 9.3.

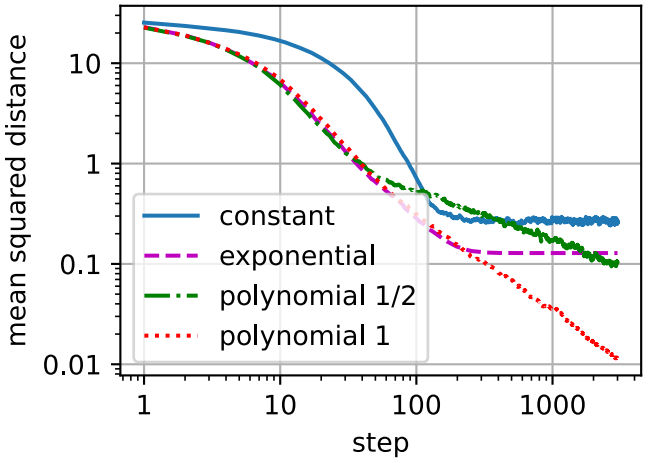
9.6.1.2 Four Schedules on the Valley

We test these predictions with 1,000 independent SGD runs on the valley, the rows of one array, all started at $(-5, -2)$ with unit gradient noise. The function `sgd_valley` returns the mean squared distance to the minimum after each step. We compare four schedules over 3,000 steps: the constant rate 0.1; exponential decay from $\eta_0 = 0.4$ with $\gamma = 0.98$; and polynomial decay from $\eta_0 = 0.4$ with $\tau = 25$ and $\alpha = 1/2$ or $\alpha = 1$. The decaying schedules start at 0.4, close to the stability limit $2/4 = 0.5$ of the steep direction.

```
def sgd_valley(schedule, num_steps=3000, num_runs=1000):
    """Mean squared distance to the minimum of SGD runs on the valley."""
    rng = np.random.default_rng()
    h = np.array(d2l.VALLEY_CURVATURES)  # Curvatures along x1 and x2
    x = np.tile([-5.0, -2.0], (num_runs, 1))
    dist = []
    for t in range(num_steps):
        x = x - schedule(t) * (h * x + rng.normal(size=x.shape))
        dist.append((x ** 2).sum(axis=1).mean())
    return dist

valley_schedules = {
    'constant': lambda t: 0.1,
    'exponential': lambda t: 0.4 * 0.98 ** t,
    'polynomial 1/2': lambda t: 0.4 / (1 + t / 25) ** 0.5,
    'polynomial 1': lambda t: 0.4 / (1 + t / 25)}
dists = {}
for name, schedule in valley_schedules.items():
    dists[name] = sgd_valley(schedule)
    budget = sum(schedule(t) for t in range(3000))
    print(f'{name:15s} budget {budget:6.1f}, final rate {schedule(2999):.2g}, '
          f'mean squared distance {dists[name][-1]:.3f}')
d2l.plot(np.arange(1, 3001), list(dists.values()), 'step',
         'mean squared distance', legend=list(dists), xscale='log',
         yscale='log')
```

```
constant        budget  300.0, final rate 0.1, mean squared distance 0.263
exponential     budget   20.0, final rate 1.9e-27, mean squared distance 0.128
polynomial 1/2  budget  200.2, final rate 0.036, mean squared distance 0.104
polynomial 1    budget   48.2, final rate 0.0033, mean squared distance 0.012
```



The three decaying schedules start at the larger rate 0.4 and approach the minimum faster than the constant rate 0.1. While the iterate is far away, every schedule makes progress; the curves part once the noise dominates. The constant rate stalls at the noise ball $\eta\sigma^2/(\lambda(2-\eta\lambda))$ of Section 9.3: summed over both coordinates it predicts a mean squared distance of 0.27, and the 1,000 runs land within a few hundredths of it. Exponential decay does better for a while, then freezes at about 0.13. Its budget, $0.4/(1-0.98) = 20$, is enough to cover the starting distance, but its rate falls by 2% per step, faster than the iterate can follow. Once $\eta_t\lambda$ drops below 0.02, the iterate no longer moves far enough to shed the noise it has collected, and that noise stays. Both polynomial schedules keep improving. With $\alpha = 1/2$ the distance falls in proportion to the rate, as the second term of Equation 9.39 predicts, although the squares of this schedule have an infinite sum. With $\alpha = 1$ it falls like $1/t$ and ends eight to nine times closer, with a budget a quarter as large: once the starting distance has been covered, the final rate decides.

That last observation carries over to deep networks, which do not train forever but for a fixed number of steps T. The conditions on infinite sums then say little. What matters are the two numbers of Equation 9.39 at the end of the run: the budget B_T, which must be large enough to cover the distance to a good solution, and the final rate η_T, which must be small enough to quench the noise. Worst-case theory for a fixed T even favors a constant rate if the average of the iterates is the output. For convex problems, SGD at a constant rate proportional to $1/\sqrt{T}$, with the average of its iterates as the output, reaches an error of order $1/\sqrt{T}$. No method that sees only noisy gradients has a better worst-case rate. For smooth nonconvex problems the same constant rate is optimal for driving the gradient toward zero (Section 35.4.3). Training, however, returns the last iterate, and for the last iterate the worst-case optimal schedule decays linearly to zero (Defazio et al. 2023), again a statement about the final rate. No theorem ranks one decay shape against another on a deep network, so we turn to experiments.

9.6.2 Decay on a Network

9.6.2.1 A Testbed

We need a network cheap enough to train many times. A LeNet-style convolutional network on Fashion-MNIST (Section 6.6) fits: two convolutional layers with 6 and 16 channels and three fully connected layers, about 62,000 parameters, trained on 60,000 images

in minibatches of 256. We use ReLU activations, max-pooling, and batch normalization (Section 7.3) after every hidden layer, and we set the initialization explicitly: Xavier (Section 4.4.2.1) for the weights and zero for the biases. Both choices matter here. Batch normalization widens the range of learning rates at which a network trains (Santurkar et al. 2018), and this section probes that range up to its edge. The edge depends on the scale of the initial weights, so we set their distribution ourselves instead of relying on a library default.

We save the network as `lenet_bn`, since Section 9.10 trains it again. The model `LeNetBN` around it takes its schedule as an argument: a function from the training time t, measured in epochs, to the learning rate. It also takes the number of minibatches per epoch, 235 for 60,000 images in minibatches of 256, so that the time advances by $1/235$ with each minibatch.

```
def lenet_bn():
    """LeNet with ReLU, max-pooling, batch normalization, Xavier weights."""
    net = nn.Sequential(
        nn.Conv2d(1, 6, kernel_size=5, padding=2), nn.BatchNorm2d(6),
        nn.ReLU(), nn.MaxPool2d(kernel_size=2, stride=2),
        nn.Conv2d(6, 16, kernel_size=5), nn.BatchNorm2d(16), nn.ReLU(),
        nn.MaxPool2d(kernel_size=2, stride=2), nn.Flatten(),
        nn.Linear(16 * 5 * 5, 120), nn.BatchNorm1d(120), nn.ReLU(),
        nn.Linear(120, 84), nn.BatchNorm1d(84), nn.ReLU(),
        nn.Linear(84, 10))
    for m in net.modules():  # Xavier weights, zero biases
        if isinstance(m, (nn.Conv2d, nn.Linear)):
            nn.init.xavier_uniform_(m.weight)
            nn.init.zeros_(m.bias)
    return net

class LeNetBN(d2l.Classifier):
    """lenet_bn trained by SGD with the learning rate schedule(t) at epoch
    t."""
    def __init__(self, schedule, num_batches):
        super().__init__()
        self.save_hyperparameters()
        self.net = lenet_bn()
```

The method `configure_optimizers` hands the schedule to plain SGD. The rate changes after every minibatch, not once per epoch.

```
@d2l.add_to_class(LeNetBN)
def configure_optimizers(self):
    optim = torch.optim.SGD(self.parameters(), lr=1.0)
    scheduler = torch.optim.lr_scheduler.LambdaLR(
        optim, lambda step: self.schedule(step / self.num_batches))
    return optim, scheduler
```

A PyTorch scheduler is a separate object. `LambdaLR` sets the learning rate to the optimizer's initial rate times the value of its function, so with an initial rate of 1 the rate is `schedule(t)`. The function receives the number of steps taken so far, which `d2l.Trainer` advances by calling `scheduler.step()` after every minibatch. `BatchNorm2d` and `BatchNorm1d` keep running aver-

ages with `momentum=0.1`, which weights the newest minibatch by 0.1. The loop at the end of `lenet_bn` replaces the default initialization of `Conv2d` and `Linear`, a uniform distribution with bound $1/\sqrt{n_{\text{in}}}$, by Xavier.

The function `train` trains a fresh `LeNetBN` with a given schedule, or continues training a model it is given. It returns the model and the trainer's history: the training loss averaged over each epoch, and the loss and accuracy on the 10,000 test images after each epoch, which `d2l.Trainer` records under the names `val_loss` and `val_acc`. Every run starts from a new random initialization and sees the training data in a new random order. A difference between two schedules means something only if it is larger than the difference between two runs of the same schedule.

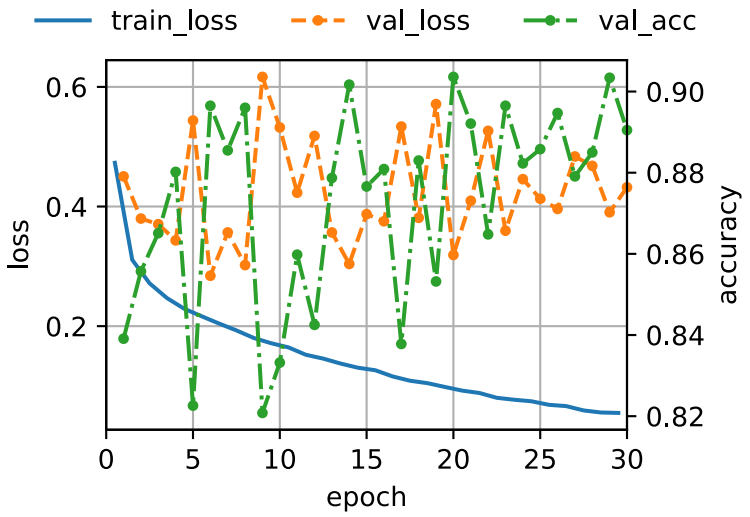
```
data = d2l.FashionMNIST(batch_size=256)
num_batches = len(data.train_dataloader())

def train(schedule, num_epochs=30, model=None, board=True):
    """Train LeNetBN under a schedule; return the model and its history."""
    if model is None:
        model = LeNetBN(schedule, num_batches)
    model.schedule = schedule
    trainer = d2l.Trainer(max_epochs=num_epochs, num_gpus=1)
    trainer.board.display = board
    trainer.fit(model, data)
    history = trainer.history
    print(f"train loss {history['train_loss'][-1]:.3f}, "
          f"test loss {history['val_loss'][-1]:.3f}, "
          f"test accuracy {history['val_acc'][-1]:.3f}")
    return model, history
```

The baseline is the schedule most training code starts with, a constant. We train for 30 epochs at $\eta = 0.3$.

```
def constant(t):
    return 0.3

_, history = train(constant)
histories = {'constant': history}
```




```
train loss 0.055, test loss 0.432, test accuracy 0.890
```

The training loss keeps falling for all 30 epochs. The test accuracy gains little after about ten epochs, and from then on it moves up and down by a point or more from one epoch to the next. The test loss drifts upward: the network fits the training images ever more closely, and the closer fit no longer carries over to the test images. The jumps of the test accuracy are the noise ball of the network: at a constant rate of 0.3 the weights never settle, and each epoch ends at a different point of the cloud.

9.6.2.2 Four Decay Shapes

We compare four decaying schedules with the constant one, all starting at $\eta_0 = 0.3$ and all running for $T = 30$ epochs. *Square-root decay*, $\eta_t = \eta_0/\sqrt{t+1}$, is the polynomial decay of Equation 9.40 with $\alpha = 1/2$ and a time scale of one epoch. *Exponential decay* multiplies the rate by $\gamma = 0.9$ per epoch. *Piecewise constant decay* divides the rate by 10 at epochs 15 and 25, the shape that trained the ImageNet classifiers of the 2010s. *Cosine decay* (Loshchilov and Hutter 2016) follows half a cosine from η_0 down to a final rate η_T,

$$\eta_t = \eta_T + \frac{\eta_0 - \eta_T}{2}\left(1 + \cos\frac{\pi t}{T}\right), \tag{9.41}$$

which glides from η_0 at $t = 0$ to η_T at $t = T$ without kinks, falls slowly at both ends and fastest in the middle, and has no milestones to choose. We let it end at $\eta_T = 0.01$.

```
def square_root(t, eta=0.3):
    return eta / math.sqrt(t + 1)

def exponential(t, eta=0.3, gamma=0.9):
    return eta * gamma ** t

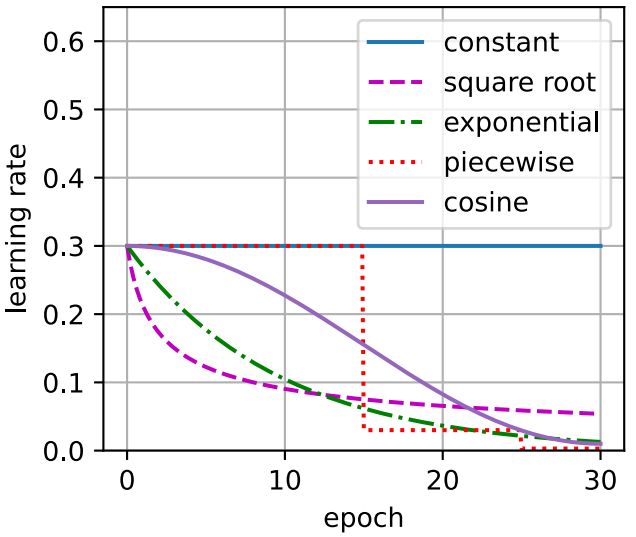
def piecewise(t, eta=0.3, milestones=(15, 25), factor=0.1):
    return eta * factor ** sum(t >= m for m in milestones)
```

```
def cosine(t, eta=0.3, eta_T=0.01, T=30):
    return eta_T + (eta - eta_T) * (1 + math.cos(math.pi * min(t, T) / T)) / 2
```

By Equation 9.39, two numbers summarize each schedule: its budget, here the area under the curve $\int_0^T \eta_t \, dt$ measured in epochs, and its final rate. We draw the five schedules and compute both.

```
schedules = {'constant': constant, 'square root': square_root,
             'exponential': exponential, 'piecewise': piecewise,
             'cosine': cosine}
ts = np.linspace(0, 30, 301)
rates = {name: np.array([float(s(t)) for t in ts])
         for name, s in schedules.items()}
for name, r in rates.items():
    budget = r[:-1].mean() * 30  # Left Riemann sum of the area
    print(f'{name:12s} budget {budget:5.2f}, final rate {r[-1]:.3f}')
d2l.plot(ts, list(rates.values()), 'epoch', 'learning rate',
         legend=list(rates), ylim=[0, 0.65], figsize=(3.5, 3))
```

```
constant     budget  9.00, final rate 0.300
square root  budget  2.75, final rate 0.054
exponential  budget  2.74, final rate 0.013
piecewise    budget  4.82, final rate 0.003
cosine       budget  4.66, final rate 0.010
```



The shapes fall into two groups. Square-root and exponential decay cut the rate early: square-root decay halves it by epoch 3 and exponential decay by epoch 7, and both end with a budget of under 3, a third of the constant schedule's 9. Piecewise constant and cosine decay keep the rate high for longer, reach budgets of about 4.8 and 4.7, and end at 0.01 or below. Square-root decay is the slowest to come down at the end; its final rate, $0.3/\sqrt{31} \approx 0.054$, is five times that of cosine decay.

9.6.2.3 Five Schedules on the Network

We train the network under each of the four decaying schedules, without drawing the progress of every run, and add the results to the constant baseline.

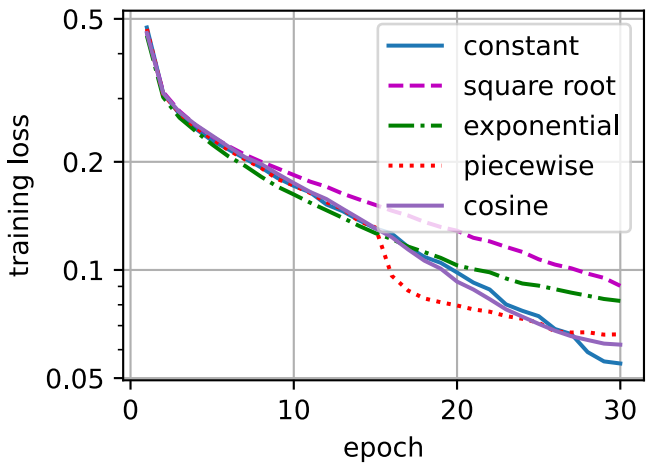
```
for name in ['square root', 'exponential', 'piecewise', 'cosine']:
    print(f'{name}: ', end='')
    _, histories[name] = train(schedules[name], board=False)
```

```
square root: train loss 0.090, test loss 0.323, test accuracy 0.897
exponential: train loss 0.082, test loss 0.289, test accuracy 0.908
piecewise: train loss 0.066, test loss 0.284, test accuracy 0.911
cosine: train loss 0.062, test loss 0.310, test accuracy 0.907
```

We draw the training loss of every run on a logarithmic scale. For the test accuracy, the table gives two numbers per run over its last five epochs: the mean, and the standard deviation from epoch to epoch, which measures how much the result depends on the epoch at which we stop.

```
epochs = np.arange(1, 31)
d2l.plot(epochs, [h['train_loss'] for h in histories.values()], 'epoch',
         'training loss', legend=list(histories), yscale='log')
print('test accuracy over the last 5 epochs: mean, std')
for name, h in histories.items():
    acc = np.array(h['val_acc'][-5:])
    print(f'{name:12s} {acc.mean():.3f} {acc.std():.3f}')
```

```
test accuracy over the last 5 epochs: mean, std
constant     0.891 0.008
square root  0.898 0.003
exponential  0.910 0.001
piecewise    0.912 0.000
cosine       0.907 0.000
```



The two groups separate on the training loss. Square-root and exponential decay end with a higher training loss than even the constant rate. Most of their budget comes early, and

with a third of the constant schedule's budget the network has not traveled as far. Piecewise constant and cosine decay drop below the constant rate as soon as their rate falls, piecewise decay abruptly at epoch 15, and they end close to it with about half its budget: their small final rate removed the noise that keeps the constant run's loss up. The constant rate catches up only in the last epochs, on twice the budget. This is Equation 9.39 at work on a network: a schedule needs both a large budget and a small final rate, and the shapes that stay high for most of the run and decay at the end provide both.

The test accuracies sort the schedules by their final rate. The three that end at a rate of about 0.01 or below, exponential, piecewise constant, and cosine decay, reach a mean of about 91% over the last five epochs, within about a point of each other, too close to rank them from single runs. The constant rate ends about two points lower, and square-root decay, whose final rate is still 0.054, lies in between. The second column shows why. As the rate falls, the test accuracy settles to a value instead of jumping from epoch to epoch: for the three schedules with a small final rate its standard deviation over the last five epochs is a few tenths of a point at most, several times smaller than for the constant rate, and the checkpoint we keep at the end is as good as any of the last few. Among the late-decaying shapes, cosine decay became widespread because it is convenient. Besides the peak rate it takes only a final rate, and it has no milestones that must be moved whenever the budget changes.

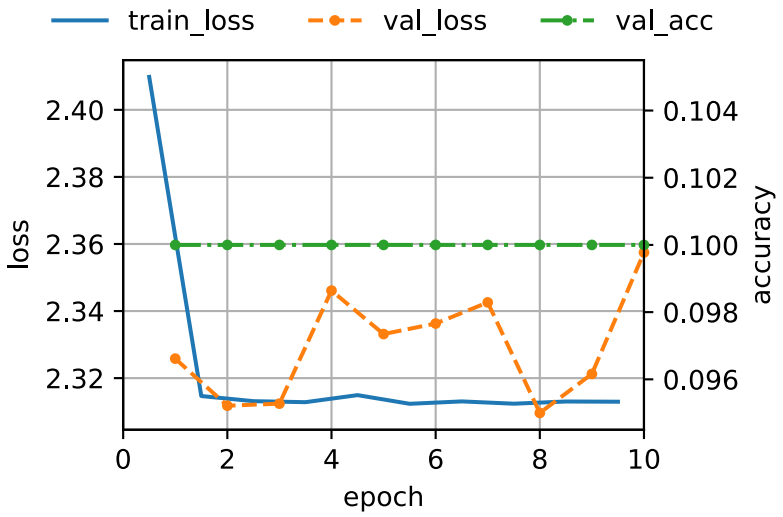
9.6.3 Warmup

Decay sets the end of the schedule. *Warmup* sets its beginning: it raises the rate from near zero to its peak over the first part of training. At initialization the weights are random, and the loss surface around them can curve sharply in some directions. A learning rate that the network tolerates after a few epochs of training can then be too large in the first steps, and the updates can push the weights into a region from which training at that rate does not recover.

Goyal et al. (2017) met this problem when they scaled ImageNet training to large mini-batches and correspondingly large learning rates, and their fix, a linear ramp of the rate over the first five epochs, became standard practice. Kalra and Barkeshli (2024) studied why it works. Early training at a growing rate reduces the sharpness of the loss around the iterate, its largest curvature, so by the time the rate reaches its peak, the network has moved to a region where that peak is stable. Adaptive optimizers such as Adam (Section 9.5) have a second, smaller reason to warm up: Adam's first update has size η in every coordinate, several times its stationary size on a coordinate whose gradient is mostly noise, until the momentum average has filled (Section 35.5.4). Our network uses plain SGD, so it isolates the first reason.

Batch normalization lets the network tolerate a wide range of learning rates, but not every rate. We train for ten epochs at a constant $\eta = 7.5$, twenty-five times the baseline:

```
_ = train(lambda t: 7.5, num_epochs=10)
```




```
train loss 2.313, test loss 2.357, test accuracy 0.100
```

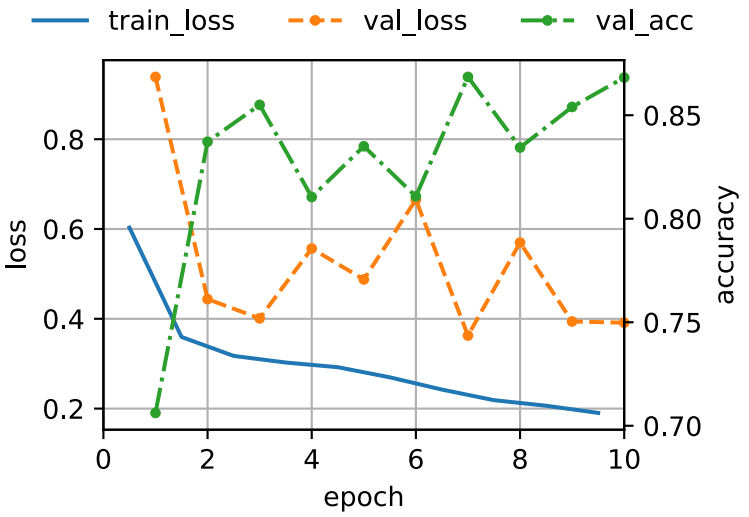
The run barely moves from chance. Its loss remains above 2, near $\log 10 \approx 2.3$, the loss of predicting all ten classes with equal probability, and the test accuracy stays below about 20%, near the 10% of guessing and far below the baseline's. A loss near $\log 10$ means that the network assigns nearly equal probabilities to the classes for every image: its output has almost stopped depending on the input.

A ramp at the start avoids this failure. The function `warmup` multiplies a schedule by a linear ramp from 0 to 1 over its first `num_epochs` epochs:

```
def warmup(schedule, num_epochs=5):
    """Ramp the rate of schedule up linearly over the first num_epochs."""
    return lambda t: min(t / num_epochs, 1) * schedule(t)
```

With a ramp of five epochs, the same target rate of 7.5 is reached at the start of epoch 6:

```
_ = train(warmup(lambda t: 7.5), num_epochs=10)
```




```
train loss 0.190, test loss 0.391, test accuracy 0.868
```

With the ramp, the network trains at the rate that failed from a cold start and reaches 80–90% test accuracy in ten epochs. Warmup does not make every rate stable: a cold start at the baseline rate of 0.3 trains, and a rate far enough above 7.5 fails even with a ramp (Exercise 5 locates both edges). The ramp widens the range of usable peak rates.

Combining the ramp with cosine decay gives a schedule that has been a common default since the late 2010s: a linear warmup to the peak, then cosine decay to near zero. It is the schedule of `d2l.cosine_warmup` (Equation 7.1), with which we trained the convolutional networks of Section 7.1 and the sections after it. That function starts the cosine where the ramp ends; here we multiply the ramp into the 30-epoch cosine schedule instead, which lowers the rate by less than 7% during the five epochs of the ramp.

```
_, histories['warmup + cosine'] = train(warmup(cosine), board=False)
```

```
train loss 0.059, test loss 0.304, test accuracy 0.908
```

9.6.4 Warmup–Stable–Decay

Cosine decay fixes the length of training, the horizon T, at the first step. Every checkpoint before T comes from a run whose rate has not yet come down. Such a checkpoint is fine for resuming training, but it is not a finished model, because its noise has not been quenched. If we decide after the fact that training should run twice as long, the cosine schedule has already reached its end, and the only clean options are to start over or to splice on a new schedule. For a single run of known length this is no limitation. It becomes one when the length is not known in advance, because data keep arriving or the loss is still falling at T, and when one run should produce models for several budgets, as in a study of scaling laws (Section 9.9).

The *warmup–stable–decay* (WSD) schedule avoids the fixed horizon (Hu et al. 2024). It warms up, holds the peak rate constant for most of the run, and decays only in a short final phase, typically the last 10–20% of the steps. Every checkpoint of the constant phase can start a

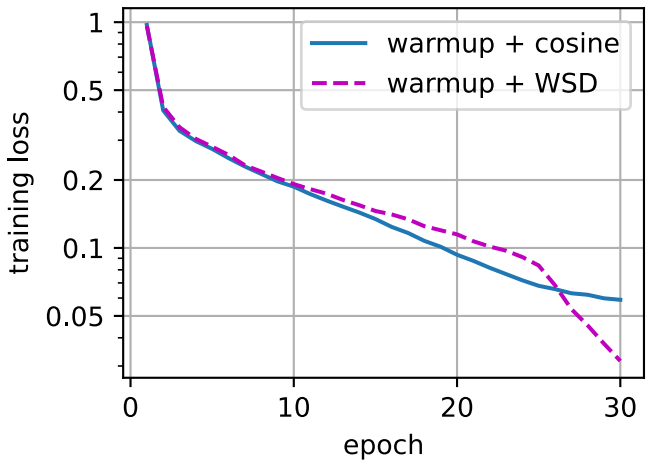
decay, so the decision of how long to train can be made late. Hägele et al. (2024) found that WSD matches cosine decay across budgets when language models are trained this way. The function `wsd` holds the rate at η until `decay_epochs` before the end and then lowers it linearly to η_T; with `warmup` in front it is the full schedule.

```
def wsd(t, eta=0.3, eta_T=0.01, T=30, decay_epochs=6):
    """Stable at eta, then a linear decay to eta_T over the last epochs."""
    progress = min(max((t - T + decay_epochs) / decay_epochs, 0), 1)
    return eta + (eta_T - eta) * progress
```

We train with a five-epoch warmup, 19 stable epochs at $\eta = 0.3$, and a decay over the last six, the same peak rate and the same 30 epochs as the warmup-plus-cosine run above, and compare the two training losses.

```
_, histories['warmup + WSD'] = train(warmup(wsd), board=False)
d2l.plot(epochs, [histories['warmup + cosine']['train_loss'],
                  histories['warmup + WSD']['train_loss']],
         'epoch', 'training loss', legend=['warmup + cosine', 'warmup + WSD'],
         yscale='log')
```

```
train loss 0.032, test loss 0.344, test accuracy 0.911
```



During the long constant phase the training loss of WSD stays above that of the cosine run, whose rate is already falling. When the decay starts at epoch 24, the WSD loss drops abruptly. The run ends at about half the training loss of the cosine run, which ran at the peak rate for a smaller part of the run, and with a test accuracy in the same range.

The same abrupt drop appears in the published loss curves of large WSD runs, and Wen et al. (2024) explain it with the geometry of a *river valley*: a loss surface with steep walls on both sides of a floor that winds gently downhill. At the high constant rate, the iterate bounces between the walls while it drifts quickly along the floor. The bouncing raises the measured loss, since the iterate is always partway up a wall, but the drift is real progress, covering a distance along the floor that small steps could not cover in the same time. The decay stops the bouncing: the iterate settles onto the part of the floor it has already reached, and the loss falls quickly.

In this model the constant phase travels along the shallow direction and the decay removes the oscillation across the steep one, which is why the constant phase can run for as long as training continues and a short decay suffices. It is the noise ball of Equation 9.39 on a winding valley instead of a bowl: the travel along the floor is the budget, and the bouncing between the walls is the noise of the current rate. Figure 9.8 draws two runs on such a valley. The same noise drives both runs, and the only difference between bouncing indefinitely and landing on the floor is what the learning rate does at the end.

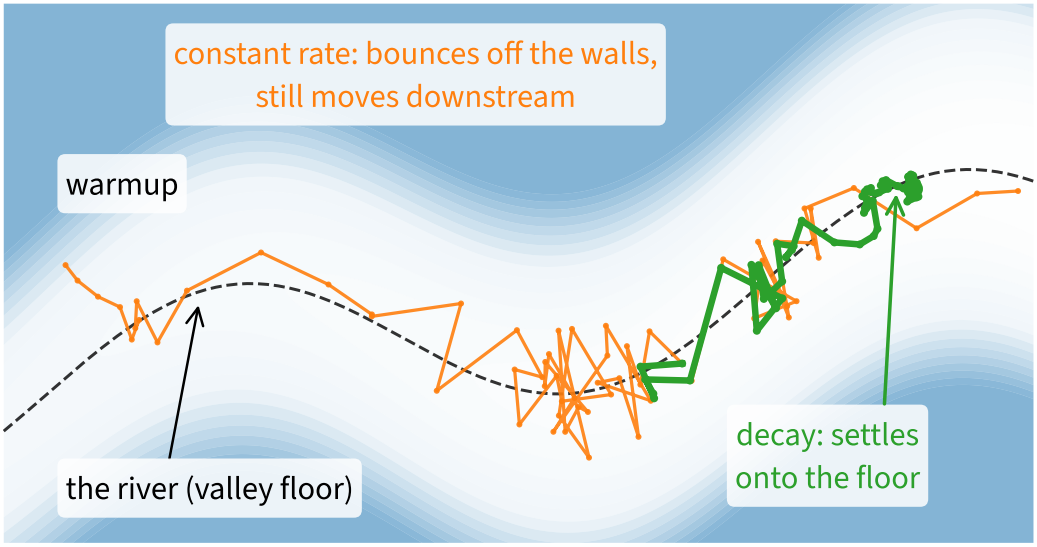


Figure 9.8: The river-valley model of warmup–stable–decay. Stochastic gradient descent on a winding valley with steep walls: after a brief warmup, the constant high learning rate bounces off the walls yet keeps moving downstream, while the branch whose learning rate decays settles onto the valley floor it had already reached.

9.6.4.1 Branching Off the Plateau

If the oscillation across the valley is what keeps a constant-rate checkpoint from being a finished model, then a decay started from *any* checkpoint of the constant phase should finish it. We test this with the workflow WSD is used for. First we train with warmup and a constant rate only, for 24 epochs. This is the run an open-ended training job would be in the middle of.

```
model, history_plateau = train(warmup(constant), num_epochs=24, board=False)
```

```
train loss 0.089, test loss 1.064, test accuracy 0.819
```

We now decide on a budget of 30 epochs. We copy the checkpoint and *branch* a six-epoch decay off the copy: it continues `warmup(wsd)` from epoch 24, the schedule that the WSD run above followed from the start. The original continues at the constant rate for the same six epochs.

```
branch = copy.deepcopy(model)
```

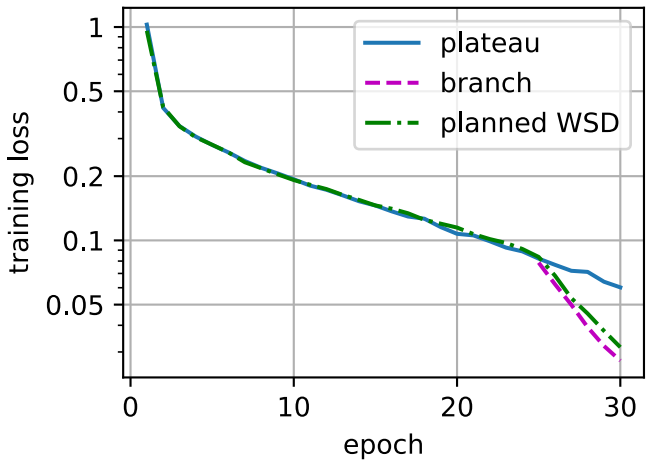
```
print('branch: ', end='')
_, history_branch = train(lambda t: warmup(wsd)(t + 24), num_epochs=6,
                          model=branch, board=False)
print('plateau: ', end='')
_, history_continued = train(constant, num_epochs=6, model=model, board=False)
```

```
branch: train loss 0.027, test loss 0.369, test accuracy 0.907
plateau: train loss 0.060, test loss 0.578, test accuracy 0.852
```

We draw the training loss of the plateau run over all 30 epochs, of the branch over its six, and of the planned WSD run of the previous experiment for comparison, and print the test accuracy of each over the last six epochs.

```
plateau = {k: history_plateau[k] + history_continued[k]
           for k in ('train_loss', 'val_acc')}
runs = {'plateau': plateau, 'branch': history_branch,
        'planned WSD': histories['warmup + WSD']}
d2l.plot([epochs, epochs[24:], epochs],
         [h['train_loss'] for h in runs.values()], 'epoch', 'training loss',
         legend=list(runs), yscale='log')
print('test accuracy, epochs 25 to 30')
for name, h in runs.items():
    print(f'{name:12s}', ' '.join(f'{a:.3f}' for a in h['val_acc'][-6:]))
```

```
test accuracy, epochs 25 to 30
plateau      0.847 0.893 0.877 0.893 0.890 0.852
branch       0.891 0.904 0.901 0.900 0.906 0.907
planned WSD  0.881 0.907 0.904 0.909 0.910 0.911
```



The training loss of the branch drops as soon as its rate decays, while the plateau run stays at the higher loss of the constant rate. After six epochs the branch ends in the same range of training loss and test accuracy as the planned WSD run, whose schedule it shares, although

nobody chose its horizon when training started. The test accuracy of the plateau run keeps jumping from epoch to epoch; at some epochs it comes close to the branch, at others it falls a point or more below. The quality of a constant-rate checkpoint depends on the epoch at which it is taken, and the decay removes that dependence.

This is the property that made WSD popular for training large models: one long constant run, with short decays branched off it whenever a model is needed, gives models at many budgets for little more than the compute of the longest one, where cosine decay needs a separate full run for each budget (Hägele et al. 2024). One practical detail: our optimizer is plain SGD, which keeps no state, so copying the parameters was enough. With momentum or Adam, the state of the optimizer must be copied along with them.

Which late-decaying shape is best remains open. In published comparisons on language models, cosine decay, WSD, and linear decay to zero end close to each other when each is well tuned, with linear decay to zero slightly ahead in recent sweeps (Hägele et al. 2024; Bergsma et al. 2025b). What is settled is a practical point: a constant phase whose checkpoints can be decayed on demand is more flexible than a schedule that fixes its horizon at the first step.

9.6.5 Summary

A learning-rate schedule is a function from training time to the learning rate. On a noisy quadratic the expected squared error splits into what remains of the starting distance, which shrinks with the budget $\sum_s \eta_s$, and the noise ball of the current rate, which shrinks with η_t. A schedule therefore needs a large budget and a small final rate. Exponential decay stops moving too early and keeps the noise it has collected; polynomial decay keeps improving. On a network trained for a fixed number of epochs, the shapes that stay high and decay late, piecewise constant and cosine, reach a lower training loss than the shapes that decay early and match the constant rate's with half its budget. The schedules that end at a small rate settle their test accuracy, about two to three points above the constant rate's. Warmup ramps the rate up at the start and widens the range of usable peak rates. Warmup–stable–decay holds the peak for most of the run and decays at the end; a decay branched off any checkpoint of its constant phase yields a finished model, so the length of training can be chosen late.

9.6.6 Exercises

1. code **The constant baseline.** Train the network of Section 9.6.2.1 with `train` at a constant $\eta \in \{0.03, 0.1, 0.3, 0.5\}$ for 30 epochs.
   1. Which rate gives the best final test accuracy, and at which epoch does each run's test accuracy stop improving?
   2. How does the gap between the test loss and the training loss at epoch 30 change as η shrinks? Relate the trend to the noise term of Equation 9.39.
2. code **Schedules on the valley.** Use the function `sgd_valley` of Section 9.6.1 for the following.
   1. Show that without noise the coordinate x_1 shrinks over a run by a factor of about $e^{-h_1 B}$, where $B = \sum_t \eta_t$ is the budget and $h_1 = 0.2$. Choose a decay factor γ for exponential decay from $\eta_0 = 0.4$ whose predicted final $|x_1|$ without noise is below 0.01, run it, and compare its final mean squared distance with that of polynomial decay with $\alpha = 1$.

   2. Run the schedule $\eta_t = c/(t+\tau)$ with $\tau = 25$ for $c \in \{1, 2.5, 10\}$ for 30,000 steps, and estimate from a log–log plot the exponent p in the decay of the mean squared distance, t^{-p}. Show that along a coordinate of curvature λ the decay is like $1/t$ only if $c\lambda > 1/2$, and explain which coordinate of the valley limits the small values of c.
   3. Piecewise constant decay with finitely many drops is not covered by the Robbins–Monro conditions. Which of the two conditions does it violate, and why is it nonetheless a sensible choice when the number of steps is fixed in advance? Use Equation 9.39 to say when the next drop should come, and test your rule on the valley with `sgd_valley`.
3. code **Polynomial decay on the network.** Train the network with the polynomial decay $\eta_t = 0.3\,(1+t)^{-\alpha}$, with t in epochs, for $\alpha \in \{0.25, 1, 2\}$, and compare with the square-root decay of Section 9.6.2.2, which is the case $\alpha = 1/2$.
   1. Compute the budget $\int_0^{30} \eta_t\, dt$ and the final rate of each of the four schedules, and relate them to the final training losses and test accuracies you measure.
   2. Polynomial decay satisfies both Robbins–Monro conditions exactly when $\alpha \in (1/2, 1]$. Does the best α on this network lie in that window? How far do conditions on infinite sums constrain a run of 30 epochs?
4. code **Schedule-free SGD.** Defazio et al. (2024) replace decay by averaging. With an interpolation weight $\beta = 0.9$ and averaging weights $c_{t+1} = 1/(t+1)$, the iteration is

$$\mathbf{y}_t = (1-\beta)\mathbf{z}_t + \beta\mathbf{x}_t, \qquad \mathbf{z}_{t+1} = \mathbf{z}_t - \eta\,\mathbf{g}_t, \qquad \mathbf{x}_{t+1} = (1-c_{t+1})\mathbf{x}_t + c_{t+1}\mathbf{z}_{t+1},$$

   where $\mathbf{g}_t$ is a stochastic gradient at $\mathbf{y}_t$, and $\mathbf{x}_t$ is the point that is evaluated.
   1. Implement it on the valley $f(\mathbf{x}) = 0.1x_1^2 + 2x_2^2$ with unit-variance Gaussian gradient noise, starting from $(-5, -2)$, and plot $f(\mathbf{z}_t)$ and $f(\mathbf{x}_t)$, averaged over 100 runs, for 3,000 steps at the constant rate $\eta = 0.1$. Compare with the constant and decaying schedules of `sgd_valley`.
   2. Explain the difference between the two curves with Equation 9.39, and state which quantity in the iteration plays the role of a decaying learning rate.
5. code **Warmup length and the stability ceiling.** ▣ The cold start at $\eta = 7.5$ in Section 9.6.3 fails, while a five-epoch ramp trains. Count a run as trained when its test accuracy after ten epochs exceeds 50%.
   1. Vary the length of the ramp in `warmup` from 0.5 to 8 epochs at $\eta = 7.5$ and report the test accuracy after ten epochs for each. What is the shortest ramp that trains?
   2. Keep the five-epoch ramp. Double the target rate from 7.5 until the ramped run no longer trains, and halve it from 7.5 until a cold start trains. By what factor does the ramp raise the largest rate that trains?
6. code **Branching at several budgets.** ▣ Continue the plateau run of Section 9.6.4.1 at the constant rate to epoch 50, and branch a six-epoch decay off it at each of epochs 30, 40, and 50. This is the design of Hägele et al. (2024) .
   1. For each branch point, record the training loss and the test accuracy of the plateau checkpoint, averaged over its last three epochs, and of the branch after its decay. Does the gain from the decay grow, shrink, or stay constant as the branch point moves later?
   2. For the branch at epoch 40, train a fresh run with `warmup(cosine)` over the same 46 epochs at the same peak rate, with the argument `T` of `cosine` set to 46. Does the branched model match it? Compare the epochs spent on the plateau run and its three branches with the epochs of three separate cosine runs at the same budgets.

7. **From noise ball to sampler.** ◻ Constant-rate SGD does not converge; its iterates fluctuate in the noise ball of the second term of Equation 9.39, a ball whose size is set by the rate. Stochastic gradient Langevin dynamics (Welling and Teh 2011) adds Gaussian noise to every step so that the iterates approximately sample the density proportional to e^{-f}:
$$\mathbf{x}_{t+1} = \mathbf{x}_t - \eta\,\mathbf{g}_t + \sqrt{2\eta}\,\zeta_t, \qquad \zeta_t \sim \mathcal{N}(\mathbf{0}, \mathbf{I}), \tag{9.42}$$
where $\mathbf{g}_t$ is a stochastic gradient of f.
   1. For $f(x) = \frac{\lambda}{2}x^2$ with the exact gradient, derive the stationary variance of x_t and compare it with the variance $1/\lambda$ of the density proportional to $e^{-f(x)}$.
   2. Compare the injected noise $\sqrt{2\eta}\,\zeta_t$ with the gradient-noise term $\eta\,\xi_t$ of SGD. For which gradient-noise variance σ^2 do the two contributions have equal variance, and which of them dominates as η decays?
8. code **Linear decay to zero.** ◻ Defazio et al. (2023) argue from a worst-case analysis that the learning rate should decay linearly to exactly zero, and Bergsma et al. (2025b) found that warmup followed by linear decay to zero matched or beat cosine and WSD in large sweeps of language-model training.
   1. Choose the arguments of `wsd` that turn `warmup(wsd)` into a five-epoch warmup to $\eta = 0.3$ followed by linear decay to zero over a 30-epoch budget, with no stable phase. Train with it and compare the final training loss and test accuracy with the `warmup(cosine)` and `warmup(wsd)` runs.
   2. Vary the final rate η_T of Equation 9.41 over $\{0.03, 0.01, 0\}$ at the same budget, and repeat the cosine run at $\eta_T = 0$ twice more. Measure the effect of the final rate by the spread of the three cosine runs, and the effect of the shape by the gap between the cosine run at $\eta_T = 0$ and the linear decay to zero. Which effect is larger on this network, and does either exceed the spread between repeated runs?

9.7 Muon

Every optimizer so far treats the parameters as one long vector of coordinates. The networks we train are not built that way. Most of their parameters sit in weight matrices that multiply activation vectors, and a step changes what each matrix does to its inputs. This section asks what the best step for such a matrix is.

In `TinyLM`, the two-block transformer language model of Section 9.5.2, 393,216 of the 470,784 parameters, about 84%, sit in eight such matrices. Adam rescales each entry of those matrices on its own and never uses the fact that they form a matrix. The gradient of such a matrix is typically dominated by a few directions (Jordan et al. 2024), so a step along the gradient moves the layer a long way along those few directions and hardly at all along the others. Adam's per-entry rescaling does not fix this, because it equalizes coordinates, not directions.

The best step for a matrix depends on how we measure the size of a step. Measured by the Euclidean norm, the best direction is the negative gradient, as in SGD. Measured by the largest change of any single coordinate, it is the sign of the gradient, the direction Adam approximates. Measured by the largest change the step can make to a layer's output, it is the gradient with all of its singular values set to one. Computing that matrix with a handful of matrix multiplications gives Muon (Jordan et al. 2024), which in under a year went from a record on a training-speed competition to the pretraining of a trillion-parameter model. We derive Muon, implement it, and compare it with AdamW on `TinyLM` and on a CNN. We

then relate it to Shampoo and look at variants that change its per-neuron step sizes and its communication cost.

```
import random
```

9.7.1 Norms and Steepest Descent

9.7.1.1 Steepest Descent Under a Ball

What does it mean for a direction to be "steepest"? Linearize the loss around the current iterate, $f(\mathbf{x}+\mathbf{d}) \approx f(\mathbf{x}) + \langle \mathbf{g}, \mathbf{d} \rangle$, and ask for the step that decreases the linearization most among all steps of a given size η:

$$\mathbf{d}^\star = \operatorname*{argmin}_{\|\mathbf{d}\| \le \eta} \langle \mathbf{g}, \mathbf{d} \rangle. \tag{9.43}$$

The problem is not posed until we say which norm defines the ball $\|\mathbf{d}\| \le \eta$, and different norms give different answers (Bernstein and Newhouse 2024). Under the Euclidean norm the ball is round, and the minimizer points straight along the negative gradient, $-\eta\,\mathbf{g}/\|\mathbf{g}\|_2$. This is the direction of SGD. SGD itself takes a step whose length grows with $\|\mathbf{g}\|_2$: it solves the penalized problem $\min_{\mathbf{d}} \langle \mathbf{g}, \mathbf{d} \rangle + \|\mathbf{d}\|_2^2/(2\eta)$ rather than the problem over a fixed ball. In this section the ball fixes the direction, and the rule for the step length is a separate choice. Following the gradient thus assumes Euclidean geometry, and each method of this chapter can be read as a different answer to the same question.

Now take the ℓ_∞ norm, $\|\mathbf{d}\|_\infty = \max_i |d_i|$. The ball becomes a box, and a step has size η as long as no single coordinate moves by more than η. To make $\langle \mathbf{g}, \mathbf{d} \rangle$ as negative as possible we push every coordinate to its wall, so the minimizer is $\mathbf{d}^\star = -\eta\,\operatorname{sign}(\mathbf{g})$: sign descent. We met this direction in Section 9.5. With both of Adam's averages switched off ($\beta_1 = \beta_2 = 0$) and $\epsilon \to 0$, Adam's update is exactly $\eta\,\operatorname{sign}(\mathbf{g}_t)$, and on language models Adam's advantage over SGD tracks the advantage of sign descent (Kunstner et al. 2023). Read geometrically, Adam is a smoothed steepest descent under ℓ_∞. The box moves every coordinate by the same amount whatever the scale of its gradient, and the moment estimates keep the signs stable under minibatch noise. This geometry suits problems whose coordinates have gradients of very different sizes, as the rare and frequent tokens of a language model do.

9.7.1.2 Matrices and the Spectral Norm

Both norms treat a parameter as an unstructured vector. A hidden matrix of `TinyLM`, however, transforms activations, $\mathbf{y} = \mathbf{W}\mathbf{x}$. For such a parameter a natural measure of a step's size is the largest change it can cause in the layer's output. That change is governed by the *spectral norm* $\|\Delta\mathbf{W}\|_2$, the largest singular value of $\Delta\mathbf{W}$:

$$\|\Delta\mathbf{W}\mathbf{x}\|_2 \le \|\Delta\mathbf{W}\|_2\,\|\mathbf{x}\|_2,$$

with equality for the worst-case input. The spectral norm is thus the largest factor by which the update can stretch any activation vector that passes through the layer. The Euclidean norm of the flattened matrix, the Frobenius norm, instead adds up the energy in all $\min(m, n)$ singular directions. It can call an update large even when its effect on every

activation is small. Controlling the spectral norm of updates is also what keeps features learning at a steady rate as networks grow wider (Yang et al. 2023).

So let the ball be spectral and solve Equation 9.43 for a matrix parameter with gradient $\mathbf{G}$. Write the reduced singular value decomposition $\mathbf{G} = \mathbf{U}\Sigma\mathbf{V}^\top$. The steepest step is the *orthogonalized gradient*:

$$\Delta\mathbf{W}^\star = \operatorname*{argmin}_{\|\Delta\mathbf{W}\|_2 \le \eta} \langle \mathbf{G}, \Delta\mathbf{W} \rangle = -\eta\, \mathbf{U}\mathbf{V}^\top. \tag{9.44}$$

The proof is one line of duality. For any $\mathbf{A}$ with $\|\mathbf{A}\|_2 \le 1$ we have $\langle \mathbf{G}, \mathbf{A} \rangle = \operatorname{tr}(\Sigma\, \mathbf{U}^\top \mathbf{A}\mathbf{V}) = \sum_i \sigma_i \, (\mathbf{U}^\top \mathbf{A}\mathbf{V})_{ii} \le \sum_i \sigma_i$, since no entry of a matrix with unit spectral norm exceeds one in absolute value. The choice $\mathbf{A} = \mathbf{U}\mathbf{V}^\top$ attains the bound, and its negative attains the minimum. (The diagonal-entry claim is an exercise. Section 35.5.5 places this result among the classical preconditioners.)

Equation Equation 9.44 equalizes singular directions. The gradient $\mathbf{U}\Sigma\mathbf{V}^\top$ names a set of directions and weights them by singular values, of which a few are large. The steepest spectral step keeps the singular vectors and replaces every nonzero singular value by one, so every direction that the gradient identifies moves at the same rate. This is per-direction equalization in the basis the gradient supplies. Adam equalizes only per coordinate, in the basis of the parameter axes.

The same reasoning tells us where the spectral norm does not apply. An embedding table never multiplies a dense activation vector. Its input is one-hot, so each row is looked up on its own, and the natural measure of a step is the largest change of any single row (Bernstein 2025). The output head and the one-dimensional vectors (biases and normalization scales) do not transform activations through a dense matrix product either. These parameters stay with AdamW. Table 9.1 summarizes the three geometries.

:Steepest descent under three norms. Each ball yields a closed-form step and suits a different group of parameters.

Table 9.1: Muon Norms

ball on the step	steepest step	suited to
Euclidean, $\|\mathbf{d}\|_2 \le \eta$	$-\eta\, \mathbf{g}/\|\mathbf{g}\|_2$	no structure assumed (SGD direction)
box, $\|\mathbf{d}\|_\infty \le \eta$	$-\eta\, \operatorname{sign}(\mathbf{g})$	coordinates of very different scale (Adam family)
spectral, $\|\Delta\mathbf{W}\|_2 \le \eta$	$-\eta\, \mathbf{U}\mathbf{V}^\top$	hidden matrices acting on activations (Muon)

Figure 9.9 draws the three choices for one gradient.

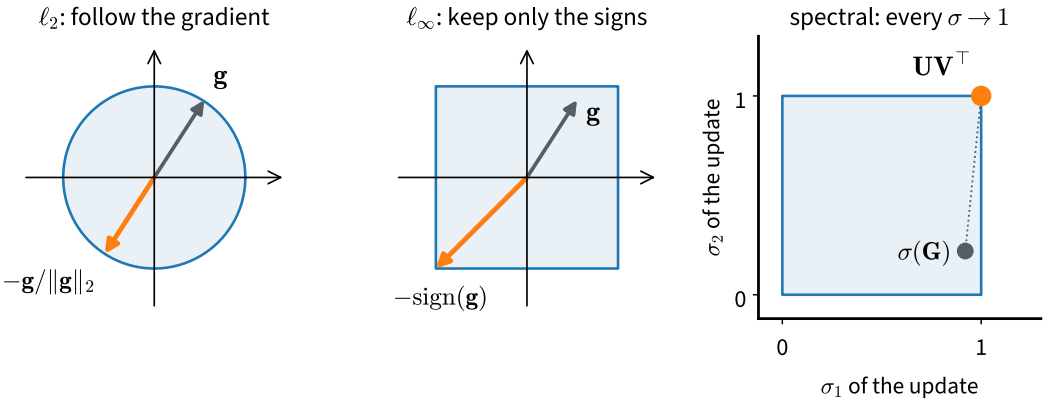


Figure 9.9: Steepest descent depends on the ball. Under ℓ_2 the best step within the ball follows the gradient, which gives normalized gradient descent. Under ℓ_∞ it moves to a corner and keeps only the signs. Under the spectral norm, drawn in singular-value coordinates where the ball is the unit square, the best matrix update sits at the corner where every singular value equals one, whatever the spectrum $\sigma(\mathbf{G})$ of the gradient: that corner is the orthogonalization $\mathbf{UV}^\top$.

9.7.2 Orthogonalization by Newton–Schulz

9.7.2.1 Odd Matrix Polynomials

Equation Equation 9.44 asks for $\mathbf{UV}^\top$, and an SVD of every matrix at every step is out of the question. An SVD is expensive, parallelizes poorly on accelerators, and is not available in the low-precision arithmetic in which training runs. Muon computes $\mathbf{UV}^\top$ approximately with matrix multiplications alone. The key fact is that an *odd* polynomial applied to a matrix acts on its singular values and leaves its singular vectors alone:

$$p(\mathbf{X}) = a\mathbf{X} + b\,(\mathbf{X}\mathbf{X}^\top)\mathbf{X} + c\,(\mathbf{X}\mathbf{X}^\top)^2\mathbf{X} = \mathbf{U}\,p(\Sigma)\,\mathbf{V}^\top. \tag{9.45}$$

Each factor $\mathbf{X}\mathbf{X}^\top$ equals $\mathbf{U}\Sigma^2\mathbf{U}^\top$, and the orthogonal factors cancel in the product. So a polynomial of the matrix is the same polynomial applied to each singular value, with $\mathbf{U}$ and $\mathbf{V}$ untouched. We can therefore drive every nonzero singular value toward 1 without ever computing one, by iterating a scalar polynomial for which 1 is an attracting fixed point. Dividing $\mathbf{X}$ by its Frobenius norm first puts every singular value in $[0, 1]$, since the Frobenius norm is at least the largest singular value. Exact zeros are fixed points of any odd polynomial, so a rank-deficient matrix stays rank-deficient, as it should.

9.7.2.2 The Tuned Quintic

Which polynomial should we iterate? A training step can afford only a few iterations, so the polynomial must move small singular values toward 1 quickly. The classical Newton–Schulz iteration uses the cubic $p(x) = \frac{3}{2}x - \frac{1}{2}x^3$. It converges to 1, but slowly from below: near zero its slope is 1.5, so a small singular value grows by only half its size per step. A singular value of 0.01 needs ten steps to pass 0.5. Jordan et al. (2024) instead tuned a quintic for speed,

$$p(x) = 3.4445\,x - 4.7750\,x^3 + 2.0315\,x^5, \tag{9.46}$$

whose slope at the origin is 3.4445. The same singular value of 0.01 passes 0.5 in four steps. Because of the steep slope, the iteration no longer converges to 1: it overshoots and oscillates in a band around it. An optimizer needs all directions to move at roughly the same rate, not orthogonality to machine precision, so the band is close enough. Figure 9.10 draws both maps, one step at a time and composed five times.

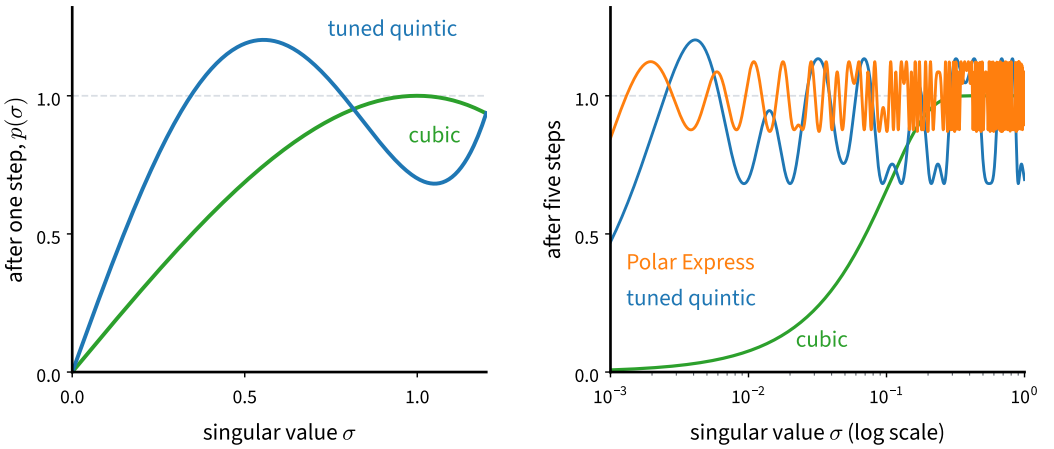


Figure 9.10: Newton–Schulz as a map on each singular value. Left: one step of the classical cubic and of the tuned quintic Equation 9.46; the quintic rises more than twice as steeply near zero and overshoots 1. Right: five steps of each, composed, for singular values from 10^{-3} to 1 on a log scale. The quintic maps every input above 0.01 into a band from about 0.68 to 1.13; the cubic brings only inputs above about 0.2 close to 1; the Polar Express polynomials of Section 9.7.2.3 map the whole range from 10^{-3} to 1 into the narrower band from 0.85 to 1.12.

The implementation below takes a list of coefficient triples (a, b, c), one per iteration, and repeats the last triple once the list runs out. Muon's choice is the single quintic Equation 9.46. A tall matrix is transposed first, so that the Gram factor $\mathbf{X}\mathbf{X}^\top$ is the smaller of the two possible squares.

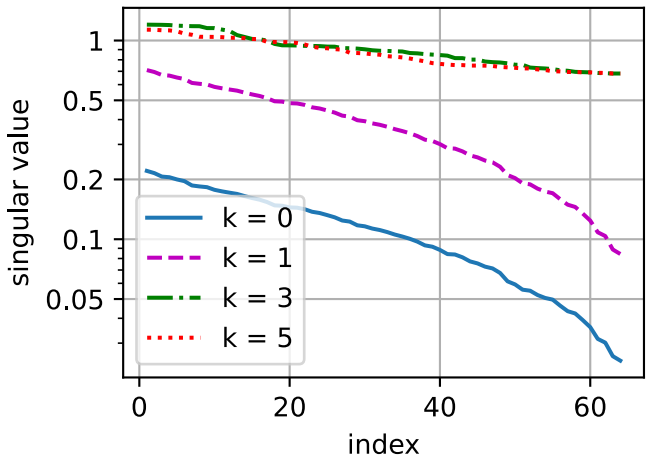
```
QUINTIC = [(3.4445, -4.7750, 2.0315)]  # Jordan et al.'s tuned quintic

def newton_schulz(M, num_iters=5, coeffs=QUINTIC, eps=1e-7):
    """Approximate the orthogonalization U V^T of M by odd polynomials."""
    tall = M.shape[0] > M.shape[1]
    X = M.T if tall else M  # Keep the Gram factor X @ X.T small
    X = X / (X.norm() + eps)
    for t in range(num_iters):
        a, b, c = coeffs[min(t, len(coeffs) - 1)]  # The last triple repeats
        A = X @ X.T
        X = a * X + (b * A + c * A @ A) @ X
    return X.T if tall else X
```

How close to orthogonal is the result? We draw a random 96×64 Gaussian matrix, whose singular values after normalization span about an order of magnitude, and plot its spectrum after 0, 1, 3, and 5 iterations. The SVD in this cell only measures the result; the iteration never computes one.

```
G = torch.randn(96, 64)
sigmas = [torch.linalg.svdvals(newton_schulz(G, num_iters=k))
          for k in (0, 1, 3, 5)]
d2l.plot(torch.arange(1, 65), sigmas, 'index', 'singular value',
         legend=[f'k = {k}' for k in (0, 1, 3, 5)], yscale='log')
print(f'singular values after 5 iterations: '
      f'[{sigmas[-1].min():.2f}, {sigmas[-1].max():.2f}]')
```

```
singular values after 5 iterations: [0.68, 1.13]
```



One iteration lifts the whole spectrum, and three compress it to within a factor of a few. After five, every singular value lies in a band around 1, from about 0.7 to 1.13, for essentially every random draw. That is $\mathbf{UV}^\top$ for an optimizer's purposes, and it takes fifteen matrix multiplications. A singular value below about 1% of the Frobenius norm, at the left end of the right panel of Figure 9.10, needs a few more iterations to climb into the band. A random Gaussian matrix has no such singular values, but we will see that real gradients have many.

How expensive are fifteen multiplications next to an elementwise Adam update? For an $m \times n$ matrix with $m \leq n$, one iteration costs $4m^2n + 2m^3$ floating-point operations: $2m^2n$ for the Gram factor, $2m^3$ for its square, and $2m^2n$ for the final product. For a square matrix that is $6m^3$, or $30m^3$ for five iterations. The forward and backward passes through the same matrix cost about $6m^2$ operations per token, or $6m^2B$ for a minibatch of B tokens. The ratio is $5m/B$ (Jordan et al. 2024). For a GPT-2-small layer ($m = 768$) and a batch of 524,288 tokens it is 0.7%. The hidden matrices of `TinyLM` are not square: each has $m = 128$ and n between 128 and 512. Counting their operations in the same way, with $6mnB$ for the forward and backward passes and a minibatch of $B = 64 \times 64 = 4{,}096$ tokens, gives about 12%. On our small model the orthogonalization is a noticeable share of the work. At scale it is nearly free.

9.7.2.3 Optimized Polynomials

The tuned quintic uses the same polynomial at every step. It cannot converge, because $p(1) = 0.701$ for the coefficients in Equation 9.46, so the iterate keeps bouncing between about 0.7 and 1.13. Nothing forces us to repeat one polynomial, however. Early steps should lift small singular values as fast as possible, and late steps should pull the whole band tight around 1. *Polar Express* (Amsel, Persson, et al. 2026) chooses a different quintic p_t for each step t. Suppose that before step t every singular value lies in an interval $[\ell_t, u_t]$, starting from $[10^{-3}, 1]$. The step-t polynomial is the odd quintic that stays closest to 1 in the worst case over that interval:

$$p_t = \operatorname*{argmin}_{p \text{ odd quintic}} \max_{x \in [\ell_t, u_t]} |1 - p(x)|, \qquad \ell_{t+1} = p_t(\ell_t), \quad u_{t+1} = 2 - \ell_{t+1}. \tag{9.47}$$

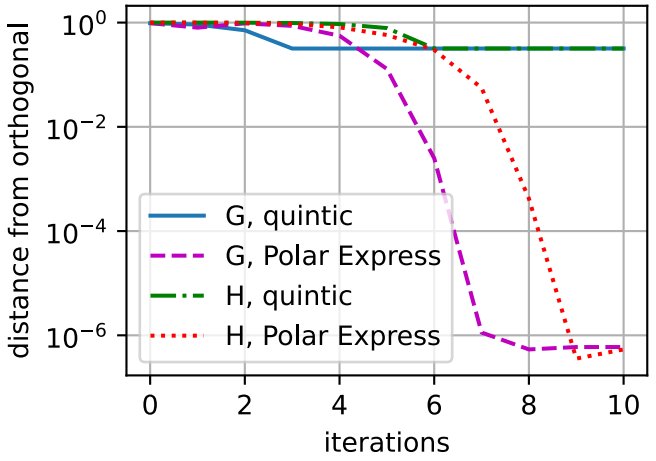
Each p_t thus maps the current interval into the narrowest possible band around 1, and that band is the interval for the next step. The update of the upper end comes from a property of minimax polynomials: the error of the optimal p_t reaches its largest value E both below 1 (at ℓ_t) and above it, so p_t maps the interval onto $[1-E, 1+E] = [\ell_{t+1}, 2-\ell_{t+1}]$. The authors prove that this greedy choice is optimal for the composition as a whole: no other sequence of T odd quintics has a smaller worst-case error after T steps. The coefficients depend only on ℓ_1, so they are computed once, offline, and hard-coded. Two practical changes make the published table differ slightly from the exact recursion. The first few polynomials are deliberately a little less aggressive than the optimal ones, which keeps the iteration stable in low precision, and each polynomial's argument is divided by 1.01 to guard against round-off in `bfloat16`. The rounded coefficients below include that factor in all but the last triple. From the seventh step on, the optimal polynomial is the classical quintic $(15x - 10x^3 + 3x^5)/8$, which converges to 1; the last triple repeats it without the safety factor.

We measure the distance of the iterate from $\mathbf{UV}^\top$ in the spectral norm, which by Equation 9.45 is $\max_i |1 - p(\sigma_i)|$. Besides the Gaussian matrix `G` we build an ill-conditioned matrix `H` with the same singular vectors and singular values spread evenly on a log scale from 1 to 10^{-3}, a condition number of 1000.

```
POLAR_EXPRESS = [(8.2052, -22.9019, 16.4607), (4.0664, -2.8612, 0.5184),
                 (3.9096, -2.8234, 0.5250), (3.2856, -2.4153, 0.4853),
                 (2.2779, -1.6198, 0.3985), (1.8726, -1.2307, 0.3585),
                 (1.8564, -1.2132, 0.3568), (1.875, -1.25, 0.375)]

def distance(M, k, coeffs):  # Spectral distance from U V^T after k steps
    sigma = torch.linalg.svdvals(newton_schulz(M, k, coeffs))
    return float((1 - sigma).abs().max())

U, _, Vt = torch.linalg.svd(G, full_matrices=False)
H = U @ torch.diag(torch.logspace(0, -3, 64)) @ Vt  # Condition number 1000
d2l.plot(list(range(11)), [[distance(M, k, c) for k in range(11)]
                           for M in (G, H) for c in (QUINTIC, POLAR_EXPRESS)],
         'iterations', 'distance from orthogonal',
         legend=['G, quintic', 'G, Polar Express', 'H, quintic',
                 'H, Polar Express'], yscale='log')
```



The tuned quintic stalls at a distance of about 0.32 on both matrices, set by the bottom of its band at 0.68. Polar Express keeps going: after five steps its distance on G is about 0.13, and after seven it reaches the round-off floor of single precision, near 10^{-6}. The plot also shows what "optimal" means here. On the well-conditioned G the tuned quintic is ahead at steps two to four, because Polar Express spends its early polynomials on lifting singular values as small as 10^{-3} and overshoots the large ones on the way. Polar Express is optimal for the worst case over $[10^{-3}, 1]$, not on every matrix, and on the ill-conditioned H, whose smallest singular values sit near that worst case, it leads at every step.

Does a more accurate polar factor train better? In GPT-2 runs of 124M and 774M parameters, Amsel, Persson, et al. (2026) find Polar Express slightly ahead of the tuned quintic in validation loss at every learning rate they tried, at the same five iterations and the same cost per step. They also find that more than six iterations, or even an exact SVD, do not lower the final loss any further, although five iterations leave the smallest singular values far from 1. The authors trace this to those small singular values: directions whose singular values lie far below the largest one barely affect training, so leaving them unconverged costs little.

9.7.3 Muon from Scratch

9.7.3.1 The Update

Muon combines momentum, Nesterov look-ahead, and approximate orthogonalization. Gradients are noisy, so we do not orthogonalize the raw gradient. We keep a momentum buffer, the leaky average $\mathbf{M}_t = \beta\,\mathbf{M}_{t-1} + \mathbf{G}_t$ of Section 9.4, with the same $\beta = 0.95$ used by its authors. We then hand Newton–Schulz the Nesterov blend $\mathbf{G}_t + \beta\mathbf{M}_t$, the gradient plus a look ahead along the freshly updated buffer, and rescale the result once per matrix shape. For an $m \times n$ matrix the update is

$$\mathbf{M}_t = \beta\,\mathbf{M}_{t-1} + \mathbf{G}_t, \qquad \mathbf{W}_{t+1} = \mathbf{W}_t - \eta \cdot 0.2\sqrt{\max(m,n)}\ \mathrm{NS}_5(\mathbf{G}_t + \beta\,\mathbf{M}_t), \tag{9.48}$$

where NS_5 denotes five steps of the tuned quintic.

The scale factor makes Muon a drop-in partner for AdamW. An orthogonalized $m \times n$ matrix has $\min(m,n)$ singular values equal to 1, hence Frobenius norm $\sqrt{\min(m,n)}$ and root-

mean-square (RMS) entry $1/\sqrt{\max(m,n)}$. Without a correction, a wide matrix would take smaller steps per entry than a square one. Multiplying by $0.2\sqrt{\max(m,n)}$ gives every matrix's update an entrywise RMS of $0.2\,\eta$, whatever its shape. For a 512×128 matrix, for example, the orthogonalized entries have RMS $1/\sqrt{512}\approx0.044$, and the factor $0.2\sqrt{512}\approx4.5$ brings them to 0.2, a typical RMS for the update of an AdamW step. With this convention, introduced for the Moonlight model, a learning rate and weight decay tuned for AdamW carry over to Muon (Jingyuan Liu et al. 2025), so the hybrid optimizer below needs a single learning rate. (The original Muon used a different shape factor, and the spectral-norm view suggests $\sqrt{d_{\text{out}}/d_{\text{in}}}$ (Bernstein 2025). The difference matters when transferring hyperparameters across widths, which Section 9.9 takes up.)

The implementation is short. `reshape` handles one wrinkle that we will need later: a convolution kernel is a matrix in disguise, with one row per output channel and one column per input channel and kernel position, so flattening it lets the same code precondition CNNs. Orthogonalization commutes with transposition, so it does not matter which of the two dimensions ends up as rows.

```
class Muon(torch.optim.Optimizer):
    """Steepest descent under the spectral norm: orthogonalized momentum."""
    def __init__(self, params, lr, momentum=0.95):
        super().__init__(params, dict(lr=lr, momentum=momentum))

    @torch.no_grad()
    def step(self):
        for group in self.param_groups:
            for p in group['params']:
                if p.grad is None:  # A parameter that received no gradient
                    continue
                state = self.state[p]
                if not state:  # First step: allocate the momentum buffer
                    state['buf'] = torch.zeros_like(p)
                buf = state['buf'].mul_(group['momentum']).add_(p.grad)
                G = p.grad + group['momentum'] * buf  # Nesterov momentum
                M = G.reshape(len(G), -1)             # Flattens conv kernels
                O = newton_schulz(M).reshape(p.shape)
                rms_scale = 0.2 * math.sqrt(max(M.shape))  # AdamW's RMS
                p.add_(O, alpha=-group['lr'] * rms_scale)
```

`Muon` subclasses `torch.optim.Optimizer`, so it plugs into any training loop that calls `zero_grad` and `step`. PyTorch stores a weight as (out, in) and a convolution kernel as (out, in, height, width), so `G.reshape(len(G), -1)` keeps one row per output channel.

9.7.3.2 Parameter Groups

Muon is an optimizer for hidden matrices only, so a real training run is a *hybrid*. We split the parameters into groups, as for weight decay in Section 9.5.4, and give each group its own optimizer. Hidden matrices go to Muon. Embeddings, the output head, and every one-dimensional tensor go to AdamW. We count both groups in `TinyLM`.

```
data = d2l.WikiText103(batch_size=64, num_steps=64)

def split_lm(model):
    hidden, rest = [], []
    for name, p in model.named_parameters():
        is_hidden = p.ndim == 2 and 'emb' not in name and 'head' not in name
        (hidden if is_hidden else rest).append(p)
    return hidden, rest

hidden, rest = split_lm(d2l.TinyLM(len(data.vocab)))
print(f'Muon:  {len(hidden):>2} tensors, '
      f'{sum(p.numel() for p in hidden):>7} parameters')
print(f'AdamW: {len(rest):>2} tensors, '
      f'{sum(p.numel() for p in rest):>7} parameters')
```

```
Muon:   8 tensors,  393216 parameters
AdamW: 22 tensors,   77568 parameters
```

The eight hidden matrices hold about 84% of the parameters. Most of the rest sits in the token embedding and the output head, one row or column for each of the 256 byte values. The split also shrinks the optimizer state. AdamW keeps two buffers per parameter, its first and second moment estimates, while Muon keeps one, the momentum buffer. With 84% of the parameters on Muon, the hybrid's state is about 58% of all-AdamW's. The factory below builds the hybrid. Because of the RMS matching in Equation 9.48, both halves share one learning rate.

```
class MultiOptimizer:
    """Apply independent optimizers to disjoint parameter groups."""
    def __init__(self, *optimizers):
        self.optimizers = optimizers

    def step(self):
        for opt in self.optimizers:
            opt.step()

    def zero_grad(self):
        for opt in self.optimizers:
            opt.zero_grad()

def muon_adamw(hidden, rest, lr):
    return MultiOptimizer(Muon(hidden, lr=lr),
                          torch.optim.AdamW(rest, lr=lr, weight_decay=0.0))
```

`MultiOptimizer` steps two independent optimizers, one per parameter group. A single `torch.optim.Optimizer` with two parameter groups cannot do this, because all of its groups share one update rule.

9.7.3.3 Comparison on the Language Model

The protocol follows Section 9.5. Each optimizer trains `TinyLM` on the bytes of WikiText-103 for 2,000 steps at a constant learning rate, on a five-point grid of learning rates from 10^{-3} to 10^{-1}. Every 250 steps `val_loss` computes the mean cross-entropy per byte over the

whole validation split, text the model never trains on, and the lowest validation loss at step 2,000 speaks for the optimizer. Scoring on held-out text keeps an optimizer from winning by memorizing the training text faster. "Tuned" in what follows means the best point of this coarse grid and nothing finer. Within each comparison every run starts from the same random initialization, drawn anew each time the notebook runs. Weight decay is off in both arms, so the only difference between them is the direction of the update on the hidden matrices. First the baseline, with every parameter on AdamW:

```
def val_loss(model):
    """Mean cross-entropy per byte over the validation split."""
    return math.log(d2l.lm_perplexity(model, data.val_dataloader()))

def run_lm(make_optimizer, lrs, seed):
    """d2l.lr_sweep of TinyLM, validated every 250 steps."""
    return d2l.lr_sweep(lambda: d2l.TinyLM(len(data.vocab)), make_optimizer,
                        data, lrs, seed, evaluate=val_loss,
                        metric='validation loss', every=250)

seed = random.getrandbits(32)  # One draw, shared by every run below
lrs = [1e-3, 3e-3, 1e-2, 3e-2, 1e-1]
adamw_lm, adamw_val = run_lm(
    lambda model, lr: torch.optim.AdamW(model.parameters(), lr,
                                        weight_decay=0.0), lrs, seed)
```

```
lr 0.001: final loss 1.660, validation loss 1.635
lr 0.003: final loss 1.586, validation loss 1.562
lr 0.01: final loss 1.651, validation loss 1.633
lr 0.03: final loss 2.752, validation loss 2.750
lr 0.1: final loss 3.154, validation loss 3.160
```

The hybrid runs over the same grid from the same initialization:

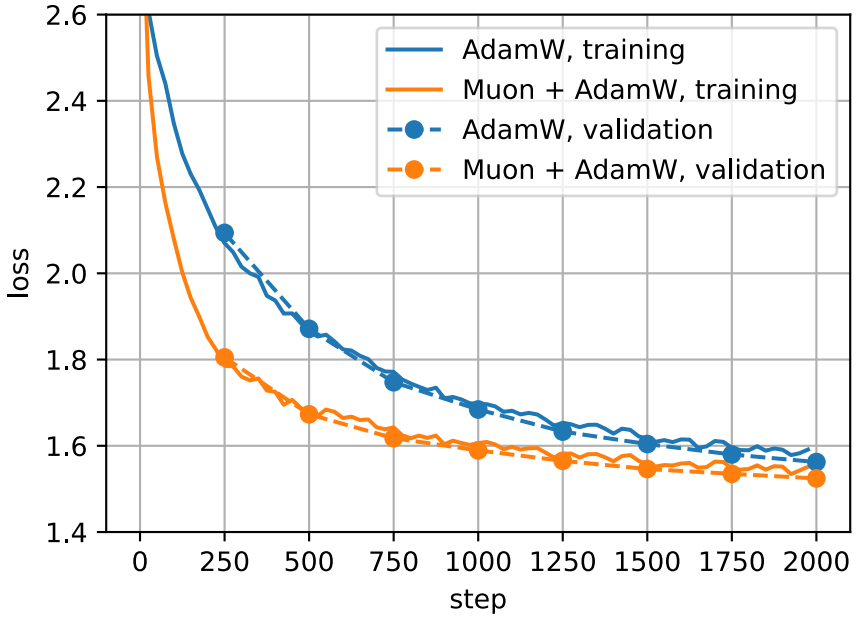
```
muon_lm, muon_val = run_lm(
    lambda model, lr: muon_adamw(*split_lm(model), lr), lrs, seed)
```

```
lr 0.001: final loss 1.613, validation loss 1.592
lr 0.003: final loss 1.549, validation loss 1.528
lr 0.01: final loss 1.545, validation loss 1.524
lr 0.03: final loss 1.663, validation loss 1.654
lr 0.1: final loss 3.053, validation loss 3.009
```

```
best_adamw = min(adamw_val, key=lambda lr: adamw_val[lr][-1])
best_muon = min(muon_val, key=lambda lr: muon_val[lr][-1])
d2l.plot([list(range(0, 2000, 25))] * 2 + [list(range(250, 2001, 250))] * 2,
         [d2l.smooth(adamw_lm[best_adamw]), d2l.smooth(muon_lm[best_muon]),
          adamw_val[best_adamw], muon_val[best_muon]], 'step', 'loss',
         legend=['AdamW, training', 'Muon + AdamW, training',
                 'AdamW, validation', 'Muon + AdamW, validation'],
         fmts=('C0-', 'C1-', 'C0o--', 'C1o--'), ylim=(1.4, 2.6),
         figsize=(5, 3.5))
for name, lr, vals in [('AdamW', best_adamw, adamw_val[best_adamw]),
```

```
                    ('Muon+AdamW', best_muon, muon_val[best_muon])]:
    print(f'{name:<10} lr {lr:g}: validation {vals[1]:.3f} at step 500, '
          f'{vals[-1]:.3f} at 2000 ({vals[-1] / math.log(2):.2f} bits)')
```

```
AdamW      lr 0.003: validation 1.871 at step 500, 1.562 at 2000 (2.25 bits)
Muon+AdamW lr 0.01: validation 1.673 at step 500, 1.524 at 2000 (2.20 bits)
```



Both optimizers have their best rate inside the grid, so the comparison is between two tuned methods. The validation loss falls at every checkpoint and stays slightly below the training loss: a run reads only a few percent of the training text and rarely sees the same text twice, so there is nothing to memorize. The tuned hybrid's validation loss is lower at every checkpoint. At step 500 it is about 1.7 nats per byte against about 1.9 for AdamW. The hybrid reaches 1.6 after 1,000 to 1,250 steps, AdamW after 1,500 to 2,000, and the hybrid matches AdamW's final validation loss after about 1,500 of the 2,000 steps. At step 2,000 the hybrid is a few hundredths of a nat ahead, at about 1.52 to 1.55 nats per byte (about 2.2 bits per byte) against about 1.56 to 1.58 for AdamW (about 2.26 bits per byte). The gap narrows as training continues but does not close, and the hybrid needs only 58% of AdamW's optimizer state.

A model of 0.47 million parameters demonstrates the mechanism but cannot rank optimizers. Wen et al. (2025) tuned ten optimizers separately at each of four model sizes, from 0.1B to 1.2B parameters. The fastest methods all precondition with matrices, Muon among them, but their speedup over a well-tuned AdamW falls from about $1.4\times$ at 0.1B parameters to about $1.1\times$ at 1.2B. The same study found no significant speedup below 0.5B parameters for Sophia, a second-order method for which an overlapping group of authors had reported $2\times$ (H. Liu et al. 2024). Claims about optimizers therefore need matched tuning and a stated model scale, and a real gain for Muon is measured in tens of percent, not in multiples.

9.7.3.4 Comparison on a CNN

A transformer is one workload. The second is the compact Fashion-MNIST CNN `d2l.FashionCNN` of Section 9.5.2.4, with two convolution and pooling stages, `conv1` and `conv2`, followed by two linear layers, `fc1` and `fc2`. The convolution kernels and `fc1` go to Muon, flattened as described above. The output layer `fc2` and the biases stay with AdamW. We record the training loss and the accuracy on the test set.

```
fashion = d2l.FashionMNIST(batch_size=256)

def split_cnn(model):
    is_hidden = lambda name, p: p.ndim >= 2 and not name.startswith('fc2')
    params = list(model.named_parameters())
    return ([p for n, p in params if is_hidden(n, p)],
            [p for n, p in params if not is_hidden(n, p)])

def run_cnn(make_optimizer, lrs, seed):
    """d2l.lr_sweep of FashionCNN, with the test accuracy at the end."""
    return d2l.lr_sweep(d2l.FashionCNN, make_optimizer, fashion, lrs, seed,
                        evaluate=lambda model: d2l.evaluate_accuracy(
                            model, fashion.val_dataloader()),
                        metric='test accuracy')

adamw_cnn, adamw_acc = run_cnn(
    lambda model, lr: torch.optim.AdamW(model.parameters(), lr,
                                        weight_decay=0.0),
    [3e-4, 1e-3, 3e-3, 1e-2], seed)
```

```
lr 0.0003: final loss 0.286, test accuracy 0.887
lr 0.001: final loss 0.201, test accuracy 0.909
lr 0.003: final loss 0.135, test accuracy 0.913
lr 0.01: final loss 0.239, test accuracy 0.891
```

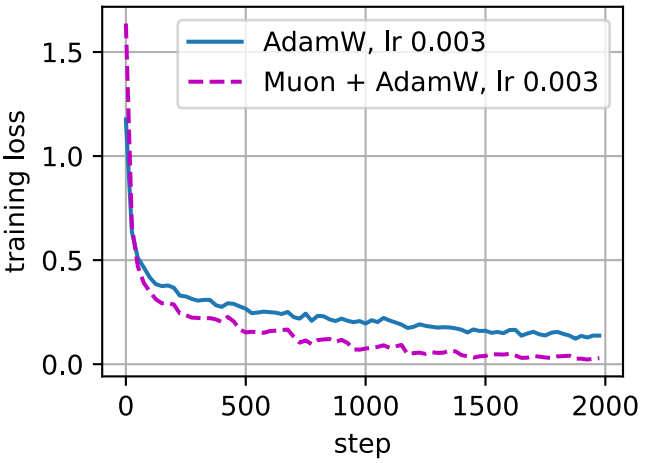
```
muon_cnn, muon_acc = run_cnn(
    lambda model, lr: muon_adamw(*split_cnn(model), lr),
    [3e-4, 1e-3, 3e-3, 1e-2], seed)
```

```
lr 0.0003: final loss 0.162, test accuracy 0.916
lr 0.001: final loss 0.034, test accuracy 0.919
lr 0.003: final loss 0.026, test accuracy 0.917
lr 0.01: final loss 0.106, test accuracy 0.907
```

```
best_adamw_cnn = min(adamw_cnn, key=lambda lr: d2l.final_loss(adamw_cnn[lr]))
best_muon_cnn = min(muon_cnn, key=lambda lr: d2l.final_loss(muon_cnn[lr]))
d2l.plot(list(range(0, 2000, 25)),
         [d2l.smooth(adamw_cnn[best_adamw_cnn]),
          d2l.smooth(muon_cnn[best_muon_cnn])],
         'step', 'training loss',
         legend=[f'AdamW, lr {best_adamw_cnn:g}',
                 f'Muon + AdamW, lr {best_muon_cnn:g}'])
print(f'test accuracy: AdamW {adamw_acc[best_adamw_cnn][-1]:.3f}, '
```

```
        f'Muon+AdamW {muon_acc[best_muon_cnn][-1]:.3f}')
```

```
test accuracy: AdamW 0.913, Muon+AdamW 0.917
```



The two metrics disagree. On training loss the hybrid descends faster and ends several times lower: orthogonalized updates drive this small CNN into memorizing its training set much sooner. On test accuracy, the metric we would deploy on, the two land within a point of each other. Optimizing faster and predicting better are different claims, and on a small task that saturates quickly the second changes little. Which of the two optimizers is better on this CNN depends on which of the two numbers we report.

9.7.3.5 Library Implementations

We rerun the tuned hybrid with the library version of Muon on the hidden matrices and the same AdamW on the rest, from the same initialization.

PyTorch has included `torch.optim.Muon` since version 2.9. Unlike our scratch version, it accepts only 2-D matrices, so convolution kernels would need reshaping. Its documentation directs embeddings, biases, and heads to AdamW. Like ours, it applies Nesterov momentum before orthogonalizing, and it offers the RMS-matched scale of Equation 9.48 as `adjust_lr_fn='match_rms_adamw'`. Its Newton–Schulz iteration runs in `bfloat16`. Its default weight decay is 0.1, so we switch it off to match the protocol above.

```
torch.manual_seed(seed)
model = d2l.TinyLM(len(data.vocab))
hidden, rest = split_lm(model)
optimizer = MultiOptimizer(
    torch.optim.Muon(hidden, lr=best_muon, momentum=0.95, nesterov=True,
                     weight_decay=0.0, adjust_lr_fn='match_rms_adamw'),
    torch.optim.AdamW(rest, lr=best_muon, weight_decay=0.0))
losses = d2l.train_steps(model, data, optimizer, 2000)
print(f'validation loss {val_loss(model):.3f}, '
      f'scratch {muon_val[best_muon][-1]:.3f}')
```

```
validation loss 1.524, scratch 1.524
```

The library version lands within about 0.02 of the scratch result, no more than the spread between runs: both compute the update Equation 9.48.

9.7.4 Relatives and Variants

9.7.4.1 Shampoo Without Memory

Muon belongs to a family of methods that precondition each weight matrix as a matrix rather than coordinate by coordinate. All of them answer the question raised in Section 9.2: gradient descent assumes round level sets, real losses have elongated ones, and some matrix should reshape the gradient accordingly. Adam estimates a *diagonal* such matrix from the gradient history. Shampoo (Gupta et al. 2018) estimates one small matrix for each side of a weight matrix, built the AdaGrad way from accumulated gradients. It preconditions each gradient matrix as $\mathbf{L}_t^{-1/4}\mathbf{G}_t\mathbf{R}_t^{-1/4}$, with $\mathbf{L}_t = \sum_{s\le t}\mathbf{G}_s\mathbf{G}_s^\top$ and $\mathbf{R}_t = \sum_{s\le t}\mathbf{G}_s^\top\mathbf{G}_s$. Muon is Shampoo with the memory removed. Switch off the accumulators, keep only the current gradient $\mathbf{G} = \mathbf{U}\Sigma\mathbf{V}^\top$, and

$$(\mathbf{G}\mathbf{G}^\top)^{-1/4}\,\mathbf{G}\,(\mathbf{G}^\top\mathbf{G})^{-1/4} = \mathbf{U}\Sigma^{-1/2}\mathbf{U}^\top\cdot\mathbf{U}\Sigma\mathbf{V}^\top\cdot\mathbf{V}\Sigma^{-1/2}\mathbf{V}^\top = \mathbf{U}\mathbf{V}^\top, \tag{9.49}$$

the orthogonalized gradient again (Bernstein and Newhouse 2024). Muon discards the preconditioner state and the matrix inverse roots, and keeps the geometry. In their place it uses the momentum buffer, which averages the gradient itself rather than its outer products. The same reading places Lion (X. Chen et al. 2023) in the sign branch: one momentum buffer and a sign, steepest descent under ℓ_∞ with even less state than Adam. You will implement it in the exercises. Section 35.5.5 derives the whole ladder, from diagonal through Kronecker to spectral preconditioners.

9.7.4.2 Per-Neuron Normalization

Muon equalizes singular directions, but it does not equalize *neurons*. A neuron is one output unit of a layer. Write the update with one row per output; then a neuron's share of the update is the norm of its row. For an exact $\mathbf{U}\mathbf{V}^\top$ of a full-rank matrix with no more rows than columns, every row has unit norm (an exercise asks you to show it). For a *tall* $m \times n$ matrix, with more output neurons than inputs, the squared row norms of $\mathbf{U}\mathbf{V}^\top$ average n/m, and individual rows can differ a lot. Five Newton–Schulz steps add a second source of spread, since small singular values have not yet reached 1. We measure both on one gradient of a freshly initialized `TinyLM`, for the four hidden matrices of its first block, and report the smallest and largest neuron norm, together with the gradient's rank and the share $\sigma_1/\|\mathbf{G}\|_F$ of its Frobenius norm that its largest singular value carries. The exact orthogonalization keeps only the nonzero singular values and sets them to 1. Newton–Schulz does the same, since it leaves zeros at zero.

```
def neuron_norms(O):  # PyTorch weights are (out, in): one row per neuron
    return O.norm(dim=1)

model = d2l.TinyLM(len(data.vocab))
X, Y = next(iter(data.train_dataloader()))
logits = model(X)
nn.functional.cross_entropy(logits.reshape(-1, logits.shape[-1]),
```

```
                              Y.reshape(-1)).backward()
blk = model.blks[0]
grads = {'qkv': blk['qkv'].weight.grad, 'proj': blk['proj'].weight.grad,
         'mlp1': blk['mlp'][0].weight.grad, 'mlp2': blk['mlp'][2].weight.grad}
for name, grad in grads.items():
    U_g, s, Vt_g = torch.linalg.svd(grad, full_matrices=False)
    r = torch.linalg.matrix_rank(grad)  # Drop the zero singular values
    exact = neuron_norms(U_g[:, :r] @ Vt_g[:r])
    ns = neuron_norms(newton_schulz(grad))
    print(f'{name:<4} {grad.shape[0]:>3} x {grad.shape[1]:<3}  rank {r:>3}  '
          f'sigma_1/norm {s[0] / grad.norm():.2f}  '
          f'exact {exact.min():.2f}-{exact.max():.2f}  '
          f'NS_5 {ns.min():.2f}-{ns.max():.2f}')
```

```
qkv  384 x 128  rank 125  sigma_1/norm 0.99  exact 0.37-0.72  NS_5 0.24-0.62
proj 128 x 128  rank 121  sigma_1/norm 0.99  exact 0.94-0.99  NS_5 0.44-0.65
mlp1 512 x 128  rank 127  sigma_1/norm 0.66  exact 0.28-0.79  NS_5 0.24-0.73
mlp2 128 x 512  rank 127  sigma_1/norm 0.94  exact 1.00-1.00  NS_5 0.72-0.87
```

The rows are the projection to queries, keys, and values (`qkv`), the attention output (`proj`), and the two MLP layers (`mlp1`, `mlp2`), with shapes written as outputs by inputs. No gradient has full rank. LayerNorm subtracts the mean of each vector, which removes one direction: at initialization, where LayerNorm adds no shift, the inputs of `qkv` and `mlp1` lie in a subspace of 127 dimensions, and so do the gradients that flow back into the outputs of `proj` and `mlp2`, because every path from the loss to the residual stream passes through a LayerNorm. Each gradient therefore has rank at most 127, and the square `proj` falls further short, at about 120. For the wide `mlp2` the missing direction is the vector of all ones, which takes the same small share from every row, so its neuron norms under exact orthogonalization all equal $\sqrt{127/128} \approx 0.996$, as close to 1 as one lost direction allows. Those of `proj` stay within about 10% of each other. Five Newton–Schulz steps leave both spread over a factor of about 1.2 to 1.7. The two tall matrices are uneven even under exact orthogonalization: their busiest neurons move about 1.7 to 3 times as far as their quietest.

After five steps the neuron norms are also well below 1, between about 0.2 and 0.9. A fresh gradient is close to rank one: for `qkv` and `proj` the largest singular value carries about 0.99 of the Frobenius norm. After the division by that norm, most of the other singular values start below 0.01, at the left end of Figure 9.10, and five steps do not lift them into the band around 1. On real gradients, then, the band from 0.7 to 1.13 holds only for the singular values above about 1% of the Frobenius norm.

NorMuon (Li et al. 2026) equalizes neurons with the tool of Section 9.5: a running average of squared updates, kept per neuron instead of per coordinate. Let $\mathbf{O}_t$ be the orthogonalized update of an $m \times n$ matrix with neurons as rows. NorMuon keeps one number per neuron,

$$v_{t,i} = \beta_2 \, v_{t-1,i} + (1-\beta_2) \, \frac{1}{n} \sum_{j=1}^{n} O_{t,ij}^2, \qquad \hat{O}_{t,ij} = \frac{O_{t,ij}}{\sqrt{v_{t,i}} + \epsilon}, \tag{9.50}$$

and rescales $\hat{\mathbf{O}}_t$ to the Frobenius norm of $\mathbf{O}_t$ before it enters Equation 9.48. Each neuron's step is thus divided by the recent RMS of that neuron's updates, as Adam divides each coordinate's step by the RMS of its gradients. The rescaling keeps Muon's overall step size, so the learning rates found above still apply. (The paper instead rescales to an entrywise RMS

of $0.2\,\eta$ directly, which differs only by the factor by which Newton–Schulz falls short of unit singular values.) The extra state is m numbers for an $m \times n$ matrix, a negligible addition to Muon's momentum buffer. NorMuon is the last step of a progression through this chapter: Adam equalizes coordinates, Muon equalizes singular directions, and NorMuon equalizes neurons on top of it.

```
def normalize_neurons(O, v, beta2=0.95, eps=1e-8):
    """Divide each neuron's update by its running RMS; keep the norm of O."""
    v = beta2 * v + (1 - beta2) * (O * O).mean(dim=1)
    O_hat = O / (v.sqrt()[:, None] + eps)
    return O_hat * (O.norm() / O_hat.norm()), v

O = newton_schulz(grads['mlp1'])  # 512 x 128, a tall matrix
O_hat, v = normalize_neurons(O, torch.zeros(O.shape[0]))
print(f'neuron norms {neuron_norms(O_hat).min():.3f}-'
      f'{neuron_norms(O_hat).max():.3f}; Frobenius norm {O.norm():.2f} '
      f'before, {O_hat.norm():.2f} after')
```

```
neuron norms 0.447-0.447; Frobenius norm 10.11 before, 10.11 after
```

On the first step the running average is proportional to each neuron's current mean square, so every neuron ends up with the same norm while the total size of the update is unchanged. Later, the average spans roughly the last $1/(1-\beta_2) = 20$ steps, so a neuron that keeps receiving large updates is slowed down, while one that moves a lot only once is slowed less: its single large step enters the average with weight $1-\beta_2 = 0.05$.

A PyTorch weight is stored as (out, in), so a neuron is a row and the average runs over `dim=1`.

How much does this help? The NorMuon paper reports that plain Muon needs 6% to 15% more steps than NorMuon to reach the same loss on GPT-2-style models of 124M and 350M parameters, and larger savings against AdamW at 1B to 5B parameters, where all optimizers shared one learning rate. A public benchmark, the modded-NanoGPT optimization track (Jordan 2026), fixes the model, the data, and the batch size, and counts the steps to a target loss over 10 to 20 runs per entry. There NorMuon and a Muon with a retuned auxiliary AdamW reach the target in the same number of steps. The idea adds almost no memory or computation, but its gain is small and depends on how carefully the baseline is tuned. AdaMuon (Si et al. 2025) instead keeps a full elementwise second moment after orthogonalization, which needs as much memory as Adam's. In the NorMuon paper's ablation, an AdaMuon-like variant (without AdaMuon's sign step) trails the per-neuron version.

9.7.4.3 Muon at Scale

AdamW updates each parameter on its own. When the parameters and optimizer state of a large model are split, or *sharded*, across many devices, each device can update its own shard without talking to the others. Newton–Schulz needs the *whole* matrix, because $\mathbf{X}\mathbf{X}^\top$ couples all of its rows. Every sharded hidden matrix must therefore be gathered on some device, orthogonalized there, and scattered back. The Moonlight implementation of Muon (Jingyuan Liu et al. 2025) gathers the momentum shards in `bfloat16`, orthogonalizes, and keeps the local slice of the result. Per parameter and step this adds 2 bytes of communication to AdamW's 8 (4 bytes to reduce-scatter the gradient and 4 to all-gather the updated parameters), so communication grows by at most a quarter. Moonlight runs the full Newton–Schulz iteration on every device of the data-parallel group, so the same work is repeated

on each. The NorMuon implementation instead assigns each matrix to one device, which orthogonalizes it and scatters the result back, so the work is spread across the devices (Li et al. 2026).

Dion (Ahn et al. 2025) avoids reassembling the matrix altogether. It replaces Newton–Schulz with a rank-r approximation of $\mathbf{U}\mathbf{V}^\top$, computed by one step of power iteration on the momentum buffer and warm-started from the previous step's result. Dion shrinks only the part of the buffer that this approximation captures; the rest stays in the buffer and is picked up in later steps. The power iteration needs only products with thin $m \times r$ and $n \times r$ matrices, so the devices exchange $O((m+n)r)$ numbers per matrix instead of mn. The authors report that a rank of a quarter to a half of the dimension matches full Muon for models with a few billion parameters, and that larger models tolerate lower ranks.

Kimi K2, a mixture-of-experts model with a trillion parameters, was pretrained with Muon on 15.5 trillion tokens (Kimi Team 2025a). In smaller test runs with plain Muon, the largest attention logits of some heads grew past 1,000, a level at which training usually becomes unstable. K2 therefore added *QK-clip*: after each optimizer step, for every attention head whose largest logit S on the batch exceeds a threshold $\tau = 100$, it multiplies that head's query and key projections by $\sqrt{\tau/S}$ each, which scales all of the head's logits by τ/S. The report describes the whole run without a single loss spike.

9.7.5 Summary

The direction of steepest descent depends on the norm that measures the step. The Euclidean ball gives the SGD direction, and the ℓ_∞ box gives sign descent, which Adam smooths. The spectral norm, which bounds the change of a layer's output, gives the orthogonalized gradient $\mathbf{U}\mathbf{V}^\top$ for hidden matrices. Muon computes it without an SVD, with five steps of a tuned quintic polynomial that uses only matrix multiplications. It applies the result to the Nesterov blend of gradient and momentum, and rescales it so that one learning rate serves both Muon and the AdamW that handles embeddings, the head, and vectors. Per-step optimal polynomials such as Polar Express make the orthogonalization more accurate at the same cost, and NorMuon adds Adam's per-unit normalization at the level of neurons. On `TinyLM` the tuned hybrid reaches a lower validation loss than tuned AdamW at every checkpoint, with under 60% of the optimizer state, and on a CNN it optimizes faster while generalizing about as well. Matched-tuning studies put Muon's gain over AdamW at tens of percent, shrinking with model size. All of these comparisons held the batch size fixed, and the next section asks how large it should be.

9.7.6 Exercises

1. **Diagonal-entry bound.** The duality argument for Equation 9.44 uses the claim that no entry of a matrix with unit spectral norm exceeds one in absolute value.
    1. Show that $\|\mathbf{A}\|_2 \le 1$ implies $|(\mathbf{U}^\top \mathbf{A}\mathbf{V})_{ii}| \le 1$ for every i.
    2. Identify when equality holds for all i simultaneously, and hence when the minimizer in Equation 9.44 is unique.
2. code **The RMS-matching factor.** The update Equation 9.48 rescales the orthogonalized matrix by $0.2\sqrt{\max(m,n)}$ so that its entries have root-mean-square size $0.2\,\eta$.
    1. `newton_schulz` leaves the singular values in a band around 1 rather than exactly at 1. Bound the ratio between the Frobenius norm of its output and $\sqrt{\min(m,n)}$

in terms of the band's endpoints, and measure the ratio on the 96×64 example of Section 9.7.2.

2. Instrument a short AdamW run of `TinyLM` to measure the entrywise RMS of its actual updates, per parameter tensor, over the first few hundred steps. Does the constant 0.2 fit the hidden matrices, the embeddings, or both?

3. code **How many iterations.** Rerun the tuned hybrid of Section 9.7.3.3 with `num_iters=1`, `3`, and `10` in `newton_schulz`, measuring the validation loss and the wall-clock time per step.

   1. Where does quality saturate, and how does the time per step grow with the iteration count? Compare the growth with the operation count of Section 9.7.2, which adds about m/B of the forward and backward cost per iteration for a square matrix.
   2. Now pass `coeffs=POLAR_EXPRESS` at five iterations. Does the more accurate polar factor of Section 9.7.2.3 change the validation loss by more than the difference between two runs of the tuned quintic from different initializations?

4. **The Polar Express guarantee.** The recursion Equation 9.47 starts from $[\ell_1, u_1] = [10^{-3}, 1]$.

   1. Apply the first three triples of `POLAR_EXPRESS` in turn to $\ell_1 = 10^{-3}$ to obtain ℓ_2, ℓ_3, and ℓ_4. How many steps of the classical cubic $\frac{3}{2}x - \frac{1}{2}x^3$ lift 10^{-3} to the same value as ℓ_4?
   2. Show that if p is an odd polynomial with $|1 - p(x)| \le \delta$ on $[\ell, u]$, then $p(\mathbf{X})$ is within δ of $\mathbf{U}\mathbf{V}^\top$ in the spectral norm for every matrix $\mathbf{X}$ whose nonzero singular values lie in $[\ell, u]$. Why does this make a worst-case design over an interval the right criterion for an optimizer that sees a different matrix at every step?

5. code **Embeddings in the Muon group.** The split in Section 9.7.3.2 keeps the token embedding table `token_emb` with AdamW. Its input is one-hot, so row i of its gradient collects contributions only from the positions where token i occurs.

   1. Let $\mathbf{M}$ be a matrix with fewer rows than columns and full row rank. Show that every row of its orthogonalization $\mathbf{U}\mathbf{V}^\top$ has unit Euclidean norm. What does this do to the update of a rare token's row of `token_emb`, compared with that of a frequent token's row?
   2. Move `token_emb` into the Muon group, keeping `pos_emb` and the output head with AdamW, and rerun the sweep of Section 9.7.3.3. Compare the best validation loss with that in `muon_val`, and explain the difference with your answer to the previous sub-item.

6. code **NorMuon.** Add the per-neuron normalization of Equation 9.50 to Muon, using `normalize_neurons` from Section 9.7.4.2 with one vector `v` per hidden matrix in the optimizer state (in PyTorch, inside `Muon.step`; in JAX, as a second `optax.GradientTransformation` chained after `scratch_muon`).

   1. Race NorMuon against Muon on `TinyLM` with the protocol of Section 9.7.3.3, three times with different initializations. Does NorMuon win in every run, and is its margin larger than the spread between runs?
   2. The tall matrices of `TinyLM` showed the largest spread of neuron norms. Apply the normalization only to them and repeat. What does the result say about where NorMuon's gain, if any, comes from?

7. code **Lion.** Implement Lion (X. Chen et al. 2023) in about six lines: with buffer $\mathbf{m}_t$, update

$$\mathbf{x}_{t+1} = \mathbf{x}_t - \eta \operatorname{sign}(\beta_1 \mathbf{m}_{t-1} + (1-\beta_1)\,\mathbf{g}_t), \qquad \mathbf{m}_t = \beta_2 \mathbf{m}_{t-1} + (1-\beta_2)\,\mathbf{g}_t,$$

with $(\beta_1, \beta_2) = (0.9, 0.99)$.

   1. Compare it with AdamW and the hybrid on `TinyLM` at matched five-point tuning, shifting Lion's grid down by a factor of three to ten relative to AdamW's, since every coordinate of its update has magnitude exactly η.
   2. Lion takes the sign of an interpolation weighted by β_1, not of its buffer $\mathbf{m}_t$. Show that the two coincide when $\beta_1 = \beta_2$. For $(\beta_1, \beta_2) = (0.9, 0.99)$, find the weight that each of them places on $\mathbf{g}_t$ and on $\mathbf{g}_{t-k}$ for $k \geq 1$, and explain how the choice $\beta_1 < \beta_2$ changes how quickly the update direction follows a change in the gradient.

8. **Damped orthogonalization.** The derivation of Equation 9.49 keeps only the current gradient $\mathbf{G}$. Let $\mathbf{G}$ be $m \times n$ with $m > n$ and rank n. Then $\mathbf{G}\mathbf{G}^\top$ is singular, and its inverse fourth root exists only as a pseudo-inverse. Shampoo (Gupta et al. 2018) starts both accumulators at $\epsilon\mathbf{I}$ with $\epsilon > 0$, so without memory its update direction becomes

$$(\mathbf{G}\mathbf{G}^\top + \epsilon\mathbf{I})^{-1/4}\,\mathbf{G}\,(\mathbf{G}^\top\mathbf{G} + \epsilon\mathbf{I})^{-1/4}. \tag{9.51}$$

   1. Show that this matrix has the singular vectors of $\mathbf{G}$, and find the function that it applies to each singular value σ.
   2. Unlike $\mathbf{U}\mathbf{V}^\top$, this update changes when $\mathbf{G}$ is multiplied by a positive constant. Find the singular values that it maps to within 1% of 1, and compare the role of ϵ with that of the ϵ in Adam's denominator $\sqrt{\hat{\mathbf{v}}_t} + \epsilon$.

9. **Adam without momentum.** ▨ With $\beta_1 = \beta_2 = 0$ and $\epsilon \to 0$, Adam's update is $\eta \operatorname{sign}(\mathbf{g}_t)$, the steepest step under the ℓ_∞ ball of Equation 9.43. With $\beta_1 = 0$ but $\beta_2 > 0$ the update is no longer a sign. Show that its magnitude in a coordinate is $\eta\,|g_t|/\sqrt{\hat{v}_t}$. Show further that its expectation is at most η when $\hat{v}_{t-1}$ equals the stationary mean of the squared gradients, and find how large a single step can become.

10. code **Spectral concentration of gradients.** ▨ Equation Equation 9.44 equalizes singular directions on the grounds that a training gradient's singular values are dominated by a few large ones, the case Jordan et al. (2024) make for orthogonalizing. Measure this concentration on `TinyLM`.

    1. For one hidden matrix, record the singular values of the Nesterov blend $\mathbf{G}_t + \beta\mathbf{M}_t$ at initialization, after 200 steps, and at the end of a 2,000-step hybrid run. Report the fraction of the squared Frobenius norm carried by the top 10% of singular values at each point.
    2. Compare your three fractions with the same fraction for an orthogonalized matrix and for a Gaussian random matrix of the same shape. Does the concentration grow or shrink over training, and do your measurements support the premise that a few large singular values dominate?

9.8 Batch Size

Most runs in this chapter so far used a batch size chosen once and never changed, such as the 64 sequences per step of `TinyLM` (Section 9.5). This section asks when a larger batch

trains faster. Doubling the batch from 64 to 128 sequences doubles the work in every step. If the run then needs half as many steps to reach the same loss, nothing is lost: the total work is unchanged, and on twice as many accelerators the run finishes in half the time. If it needs as many steps as before, half of the work was spent for nothing. Which of the two happens decides how fast a model can be trained, because a run can be shared among only as many accelerators as its batch can feed.

Two facts from earlier sections frame the question. Averaging b examples divides the variance of a gradient estimate by b (Section 9.3), and an accelerator processes a larger batch in barely more time per step until it is fully occupied. A larger batch therefore gives a better gradient at little extra cost per step. It saves steps, however, only as long as the noise in the gradient was what slowed the optimizer down.

To turn that observation into numbers, we compare the size of the noise in a gradient with the size of the gradient itself, measure the ratio on two small models, and use it to predict how the number of steps to a target loss falls as the batch grows. We then ask how the learning rate must change when the batch does, and why the best batch size grows while a model trains.

```
import random
```

9.8.1 The Gradient-Noise Scale

9.8.1.1 Signal and Noise

Whether a larger batch saves steps depends on how large the noise in a minibatch gradient is compared with the gradient itself. We first need a number for each. Write ∇f for the gradient of the training loss over the whole dataset at the current parameters, and ∇f_i for the gradient of the loss on example i. The per-example gradients scatter around their mean ∇f. Their expected squared distance from it,

$$\operatorname{tr}\Sigma = \mathbb{E}_i \|\nabla f_i - \nabla f\|^2, \tag{9.52}$$

is the *noise power* of one example, the trace of the covariance matrix Σ of the per-example gradients. A minibatch gradient $\hat{\mathbf{g}}_b$ averages b examples drawn independently with replacement. It is unbiased, and its noise power is b times smaller, because the variance of an average of b independent terms is $1/b$ times the variance of one:

$$\mathbb{E}\|\hat{\mathbf{g}}_b - \nabla f\|^2 = \frac{\operatorname{tr}\Sigma}{b}. \tag{9.53}$$

This is the $1/b$ law that Section 9.3 checked on a small network. It splits every minibatch gradient into a *signal* ∇f, whose squared length $\|\nabla f\|^2$ is the same at every batch size, and a *noise* whose squared length averages $\operatorname{tr}\Sigma/b$ and shrinks as the batch grows. The two have equal power at the batch size

$$b_{\text{noise}} = \frac{\operatorname{tr}\Sigma}{\|\nabla f\|^2}, \tag{9.54}$$

called the *gradient-noise scale* (McCandlish et al. 2018). If, say, $\operatorname{tr}\Sigma = 100$ and $\|\nabla f\|^2 = 1$, the gradient of a single example carries a hundred times more noise power than signal, and a batch of $b_{\text{noise}} = 100$ examples brings the two to parity.

Below b_{noise} the noise dominates. Doubling the batch halves the noise power and gives a markedly better direction, so each step can make more progress and fewer steps are needed. Above b_{noise} the signal dominates, the direction is already nearly right, and further averaging removes little error. The noise scale therefore predicts that growing the batch up to about b_{noise} reduces the number of steps without much increase in total work, and that beyond it the total work grows faster than the number of steps falls. The rest of this section tests that prediction.

9.8.1.2 A Two-Batch Estimator

Neither $\operatorname{tr}\Sigma$ nor $\|\nabla f\|^2$ can be read off a single minibatch, but both follow from a quantity that can: the squared norm of a minibatch gradient. We expand $\|\hat{\mathbf{g}}_b\|^2 = \|\nabla f + (\hat{\mathbf{g}}_b - \nabla f)\|^2$ and take expectations. The cross term vanishes because the deviation has mean zero, which leaves

$$\mathbb{E}\left[\|\hat{\mathbf{g}}_b\|^2\right] = \|\nabla f\|^2 + \frac{\operatorname{tr}\Sigma}{b}. \tag{9.55}$$

On average a minibatch gradient is longer than the true gradient, and the excess squared length is its noise power. Let N_b denote the mean squared norm of minibatch gradients at batch size b. Measuring it at a small batch size b_{s} and a large one b_{l} gives two linear equations in the two unknowns, with the solution

$$\operatorname{tr}\Sigma \approx \frac{N_{b_{\text{s}}} - N_{b_{\text{l}}}}{1/b_{\text{s}} - 1/b_{\text{l}}}, \qquad \|\nabla f\|^2 \approx \frac{b_{\text{l}} N_{b_{\text{l}}} - b_{\text{s}} N_{b_{\text{s}}}}{b_{\text{l}} - b_{\text{s}}}.$$

The difference of the two mean norms isolates the noise, and the large batch, nearly free of noise, pins down the signal. Checking the $1/b$ law directly requires the full-dataset gradient as ground truth. This estimator needs no full gradient, which makes it cheap enough to log throughout a large training run, where every data-parallel worker computes a small-batch gradient anyway and their average is the large-batch gradient.

We measure the two models of Section 9.5.2. `TinyLM` is a two-block decoder-only transformer with about 0.47 million parameters that predicts the next byte of WikiText-103 text from windows of 64 bytes. `FashionCNN` classifies Fashion-MNIST images with two convolution-and-pooling stages and two fully connected layers, with about 0.4 million parameters. For each model a sampling function draws a random minibatch of b examples. The Fashion-MNIST images sit on the device as one tensor, indexed at random. For the language model one example is a window of 64 bytes at a random offset in the training text, so its noise scale and its batch sizes count sequences, each of 64 tokens. Two windows drawn from 270 million bytes almost never overlap, so the examples of a minibatch are close to the independent draws that the estimator assumes.

```
device = d2l.try_gpu()
data = d2l.WikiText103(batch_size=64, num_steps=64)
fashion = d2l.FashionMNIST(batch_size=256)
```

```
X_f = (fashion.train.data.float() / 255).unsqueeze(1).to(device)
y_f = fashion.train.targets.to(device)

def sample_fashion(b):  # b random training images
    i = torch.randint(len(y_f), (b,), device=device)
    return X_f[i], y_f[i]

def sample_lm(b):  # b windows at random offsets in the training text
    starts = torch.randint(len(data.train) - data.num_steps, (b,))
    X, Y = data.windows(data.train, starts.numpy())
    return X.to(device), Y.to(device)
```

The estimator follows the two formulas line by line: average the squared gradient norm over many small minibatches and a few large ones, each drawn by `sample`, solve for the noise power and the signal power, and return their ratio.

```
def ce_loss(model, X, Y):
    logits = model(X)
    return F.cross_entropy(logits.reshape(-1, logits.shape[-1]),
                           Y.reshape(-1))

def grad_sq_norm(model, X, Y):
    grads = torch.autograd.grad(ce_loss(model, X, Y),
                                list(model.parameters()))
    return sum((g ** 2).sum() for g in grads)

def noise_scale(model, sample, b_small=16, b_big=2048, m=400):
    def mean_sq_norm(b, m):  # N_b, averaged over m random minibatches
        return torch.stack([grad_sq_norm(model, *sample(b))
                            for _ in range(m)]).mean()
    n_small, n_big = mean_sq_norm(b_small, m), mean_sq_norm(b_big, m // 8)
    tr_sigma = (n_small - n_big) / (1 / b_small - 1 / b_big)
    sq_norm = (b_big * n_big - b_small * n_small) / (b_big - b_small)
    return (tr_sigma / sq_norm).item()
```

The noise scale depends on the parameters, so we measure it twice: at initialization, and after 500 steps of Adam with `d2l.train_steps` from Section 9.5. That function streams minibatches and minimizes the cross-entropy of a model's outputs for a fixed number of steps, so it trains the CNN as well as the language model. First the CNN:

```
cnn = d2l.FashionCNN().to(device)
print(f'noise scale at initialization: '
      f'{noise_scale(cnn, sample_fashion):.0f}')
optimizer = torch.optim.Adam(cnn.parameters(), lr=0.004)
d2l.train_steps(cnn, fashion, optimizer, num_steps=500)
print(f'after 500 steps of Adam: {noise_scale(cnn, sample_fashion):.0f}')
```

```
noise scale at initialization: 91
after 500 steps of Adam: 226
```

Then the language model, with smaller probe batches because one example is already a sequence of 64 bytes:

```
lm = d2l.TinyLM(len(data.vocab)).to(device)
print(f'noise scale at initialization: '
      f'{noise_scale(lm, sample_lm, b_small=8, b_big=256):.1f}')
optimizer = torch.optim.Adam(lm.parameters(), lr=0.003)
d2l.train_steps(lm, data, optimizer, num_steps=500)
print(f'after 500 steps of Adam: '
      f'{noise_scale(lm, sample_lm, b_small=8, b_big=256):.0f}')
```

```
noise scale at initialization: 2.2
after 500 steps of Adam: 183
```

At initialization the noise scale is about a hundred images for the CNN and about two sequences for the language model. Both models are small, and so are their noise scales: the batches of 256 images and 64 sequences that train them elsewhere in this chapter lie above these values.

The noise scale also changes, and quickly. After 500 steps that of the language model has grown by more than an order of magnitude, to between about 50 and 200 sequences across our runs. That of the CNN has grown in most runs, but by very different amounts, and lies anywhere from several dozen to over a thousand images. At initialization the full gradient is large compared with the disagreement between examples. As training removes the easy part of the loss, $\|\nabla f\|^2$ shrinks, while the scatter of the per-example gradients shrinks more slowly. We return to this growth at the end of the section.

The estimate itself is repeatable. Measured again on the same trained model, it changes by less than a third, so the spread between runs comes from the runs themselves: after 500 steps of Adam, CNNs trained from different seeds sit at points with quite different noise scales. To choose a batch size, a value known to within a factor of two is enough. The magnitudes do not carry over to other problems: Kaplan et al. (2020) found critical batch sizes, which track the noise scale, of around a million tokens for large language models, against several thousand bytes here. What carries over is the concept and its growth during training.

9.8.2 Steps to a Target

The noise scale makes a prediction about training speed that could fail, so we test it. The test, standard in the study of large-batch training (Shallue et al. 2019), fixes a target loss, trains at a range of batch sizes, and counts the *steps* $S(b)$ that each run needs to reach the target and the *examples* $E(b) = b\,S(b)$ that it processes on the way. If doubling b halves S, the example count stays constant; we call this *perfect scaling*. It cannot continue forever. No batch, however large, gives a better gradient than the exact one, so the step count approaches a floor $S_{\min}$, the number of steps that training with the full gradient would need. A model of noisy optimization on a quadratic (McCandlish et al. 2018) connects the two limits:

$$\frac{S(b)}{S_{\min}} = 1 + \frac{b_{\text{noise}}}{b}, \qquad \frac{E(b)}{E_{\min}} = 1 + \frac{b}{b_{\text{noise}}}, \tag{9.56}$$

where $E_{\min}$ is the number of examples that a run with very small batches consumes, and b_{noise} is the noise scale averaged over the run. The step count falls like $1/b$ while b is small compared with b_{noise} and levels off at $S_{\min}$ beyond it; the example count does the reverse.

Because $E = b\,S$, the two minima are tied by $E_{\min} = b_{\text{noise}} S_{\min}$. With $b_{\text{noise}} = 100$, for instance, a batch of 10 needs 11 $S_{\min}$ steps but only 1.1 $E_{\min}$ examples, and a batch of 1,000 needs 1.1 $S_{\min}$ steps but 11 $E_{\min}$ examples. At $b = b_{\text{noise}}$ both stand at twice their minimum, and the batch size of this bend is called the *critical batch size*. Below it, most of the extra work per step turns into fewer steps; above it, most of the extra work is wasted. In the model, the noise scale weights each direction of parameter space by its curvature. The unweighted ratio Equation 9.54 that we measured agrees with it when the curvature is the same in every direction, and serves as a cheap stand-in otherwise.

Our protocol is as follows. The target is a fixed loss, evaluated every ten steps on a fixed set of examples: the first 8,192 training images for the CNN, and for the language model 2,048 windows spread over the whole validation split, text that no run trains on. Evaluating the loss of the current minibatch instead would let the small-batch runs, whose loss estimates fluctuate most, cross the target early by chance. The optimizer is Adam. Its learning rate is tuned at an anchor batch size b_0 (64 sequences for the language model, 256 images for the CNN) and moved to the other batch sizes by the square-root rule $\eta(b) = \eta_0\sqrt{b/b_0}$, which the next part of this section derives and tests. Each sweep draws one random seed and starts all its batch sizes from the same initialization, so that the differences along a curve come from the batch size and not from the draw. The step counts still vary from run to run, and we read only features of the curves that recur across runs. `eval_loss` computes the loss on the fixed examples, in chunks that bound the memory it needs.

```
def eval_loss(model, X, Y, chunk=4096):
    with torch.inference_mode():
        return torch.stack([ce_loss(model, X[i:i + chunk], Y[i:i + chunk])
                            for i in range(0, len(Y), chunk)]).mean().item()
```

The training loop is `d2l.train_steps`, stopped by the function `stop` once the target is reached:

```
def train_to_target(model, data, optimizer, X_eval, Y_eval, target,
                    max_steps, eval_every=10):
    """Return the number of steps until the eval loss reaches target."""
    def reached(step):
        return (step % eval_every == 0
                and eval_loss(model, X_eval, Y_eval) <= target)
    steps = len(d2l.train_steps(model, data, optimizer, max_steps,
                                stop=reached))
    return steps if reached(steps) else float('inf')
```

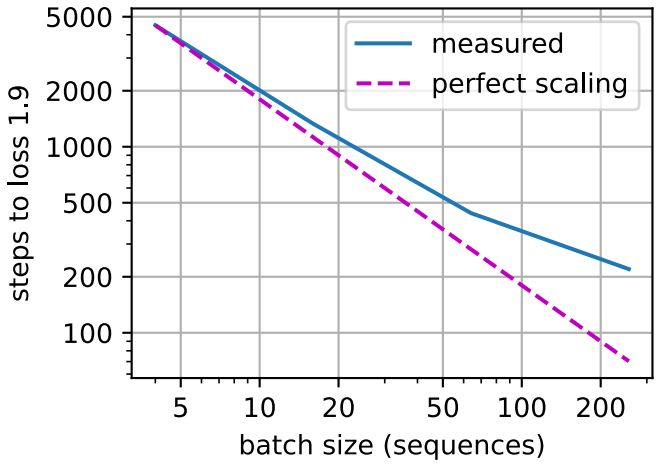
9.8.2.1 The Language Model

For `TinyLM` the target is a held-out loss of 1.9 nats per byte (2.7 bits per byte), a perplexity of $e^{1.9} \approx 6.7$ against 256 for uniform guessing over the bytes: well into learning, and far from converged. The batch sizes span two orders of magnitude, from 4 to 256 sequences. The data is built once, and only its batch size changes from run to run.

```
starts = np.arange(0, len(data.val) - data.num_steps, data.num_steps)
X_val, Y_val = (t.to(device) for t in data.windows(data.val,
                                                   starts[::8][:2048]))
```

```
lm_bs, lm_steps = [4, 16, 64, 256], []
seed = random.getrandbits(32)  # One initialization for all batch sizes
for b in lm_bs:
    torch.manual_seed(seed)
    data.batch_size = b
    model = d2l.TinyLM(len(data.vocab)).to(device)
    optimizer = torch.optim.Adam(model.parameters(),
                                 lr=0.003 * math.sqrt(b / 64))
    lm_steps.append(train_to_target(model, data, optimizer, X_val, Y_val,
                                    target=1.9, max_steps=8000))
    print(f'b={b:4d}  steps={lm_steps[-1]:6.0f}  '
          f'examples={b * lm_steps[-1]:8.0f}')
d2l.plot(lm_bs, [lm_steps, [lm_steps[0] * lm_bs[0] / b for b in lm_bs]],
         'batch size (sequences)', 'steps to loss 1.9', xscale='log',
         yscale='log', legend=['measured', 'perfect scaling'])
```

```
b=   4  steps=  4500  examples=   18000
b=  16  steps=  1330  examples=   21280
b=  64  steps=   440  examples=   28160
b= 256  steps=   220  examples=   56320
```



The first quadrupling of the batch, from 4 to 16 sequences, runs nearly parallel to the perfect-scaling line: it cuts the step count about three- to nearly fourfold, and the example count grows by at most about 40%. After it the curve bends away from the line. The last quadrupling, from 64 to 256 sequences, cuts the step count only about in half, and the run consumes three to five times as many examples as the run with the smallest batch.

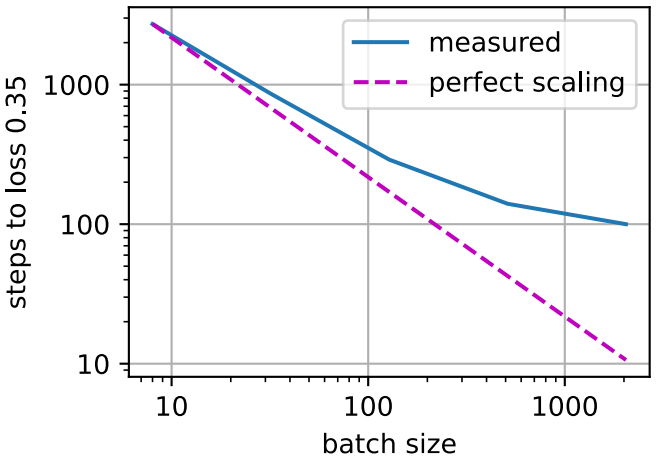
9.8.2.2 The CNN

The same experiment on the CNN, with a target loss of 0.35 and batch sizes from 8 to 2,048 images. The square-root rule sets the learning rate up to 512 images, and above that we hold it fixed. At 2,048 images the rule would double the rate to about 0.011, and there it fails in the way that Section 9.8.3.3 describes for the first steps of training: Adam's first steps overshoot, the training loss jumps above its starting value of $\log 10 \approx 2.3$, and in some runs the network falls back to uniform guessing for a few steps and trains more slowly from then

on. Such a run can need more steps than the run at 512 images. At the rate of $b = 512$ the first steps stay stable and the large batch reaches the target sooner:

```
cnn_bs, cnn_steps = [8, 32, 128, 512, 2048], []
seed = random.getrandbits(32)  # One initialization for all batch sizes
for b in cnn_bs:
    torch.manual_seed(seed)
    fashion_b = d2l.FashionMNIST(batch_size=b)
    model = d2l.FashionCNN().to(device)
    lr = 0.004 * math.sqrt(min(b, 512) / 256)  # Held fixed above 512
    optimizer = torch.optim.Adam(model.parameters(), lr=lr)
    cnn_steps.append(train_to_target(model, fashion_b, optimizer,
                                     X_f[:8192], y_f[:8192], target=0.35,
                                     max_steps=8000))
    print(f'b={b:4d}  steps={cnn_steps[-1]:6.0f}  '
          f'examples={b * cnn_steps[-1]:8.0f}')
d2l.plot(cnn_bs, [cnn_steps,
                  [cnn_steps[0] * cnn_bs[0] / b for b in cnn_bs]],
         'batch size', 'steps to loss 0.35', xscale='log', yscale='log',
         legend=['measured', 'perfect scaling'])
```

```
b=   8  steps=  2720  examples=   21760
b=  32  steps=   860  examples=   27520
b= 128  steps=   290  examples=   37120
b= 512  steps=   140  examples=   71680
b=2048  steps=   100  examples=  204800
```



The CNN shows the same pattern. The first quadrupling, from 8 to 32 images, stays close to the perfect-scaling line, cutting the step count about three- to fourfold, and after it the gap widens. At the top of the range a further quadrupling of the batch, and with it of the work per step, saves at most about half of the steps, and the run consumes two to four times the examples of the run at 512 images, six to twelve times the small-batch value. Past this point, extra work per step saves few steps and multiplies the data consumed.

9.8.2.3 The Critical Batch Size

Both curves have the shape that Equation 9.56 predicts, and we can fit the model to them. Its step count $S(b) = S_{\min} + S_{\min} b_{\text{crit}}/b$ is linear in $1/b$, so a straight-line fit of the measured step counts against $1/b$ returns $S_{\min}$ as the intercept and b_{crit} as the ratio of slope to intercept. We write b_{crit} for this fitted critical batch size, to keep it apart from the noise scale measured from gradients. The fit weights each run by the inverse of its step count, so that it matches every point to the same relative precision rather than favoring the small-batch runs with their large step counts.

```
def fit_steps(bs, steps):
    """Fit S(b) = S_min (1 + b_crit / b) by least squares in 1/b."""
    b, S = np.array(bs, dtype=float), np.array(steps, dtype=float)
    ok = np.isfinite(S)  # Leave out runs that never reached the target
    slope, S_min = np.polyfit(1 / b[ok], S[ok], 1, w=1 / S[ok])
    assert S_min > 0, 'step counts never level off: no floor to fit'
    return S_min, slope / S_min

fits = {'TinyLM': fit_steps(lm_bs, lm_steps),
        'FashionCNN': fit_steps(cnn_bs, cnn_steps)}
for name, (S_min, b_crit) in fits.items():
    print(f'{name:10s}  S_min = {S_min:4.0f} steps  b_crit = {b_crit:4.0f}')
```

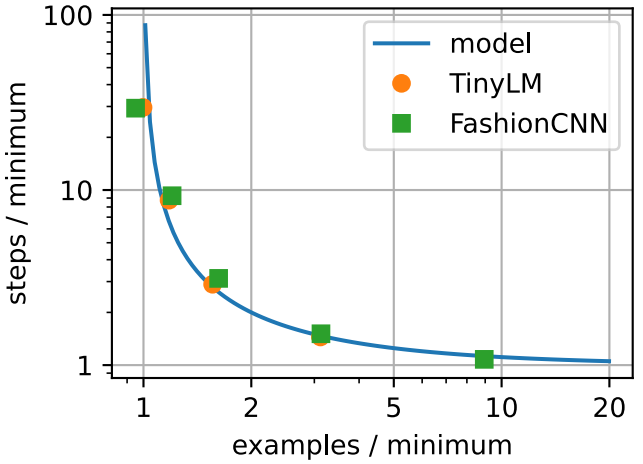
```
TinyLM      S_min =  152 steps  b_crit =  119
FashionCNN  S_min =   93 steps  b_crit =  247
```

The fitted critical batch sizes are roughly a hundred sequences for `TinyLM`, between about 70 and 120 across our runs, and two to four hundred images for the CNN, of the same order as the noise scales that each model reached during its first 500 steps. Closer agreement is more than Equation 9.56 promises here: the model uses the curvature-weighted noise scale averaged over the whole run, while we measured the unweighted ratio at two moments only.

Dividing each run's steps by $S_{\min}$ and its examples by $E_{\min} = b_{\text{crit}} S_{\min}$ removes the two fitted constants. Multiplying the two equations of Equation 9.56 after subtracting 1 from each shows that, if the model holds, every run of both models lies on the single curve

$$\left(\frac{E}{E_{\min}} - 1\right)\left(\frac{S}{S_{\min}} - 1\right) = 1.$$

```
x = np.logspace(0.005, 1.3, 100)  # E / E_min along the model curve
xs, ys = [x], [1 + 1 / (x - 1)]
for (S_min, b_crit), bs, steps in zip(fits.values(), (lm_bs, cnn_bs),
                                      (lm_steps, cnn_steps)):
    xs.append([b * S / (b_crit * S_min) for b, S in zip(bs, steps)])
    ys.append([S / S_min for S in steps])
d2l.plot(xs, ys, 'examples / minimum', 'steps / minimum', xscale='log',
         yscale='log', fmts=('-', 'o', 's'),
         legend=['model', 'TinyLM', 'FashionCNN'])
```



Part of this agreement is built in, because each model's two constants were fitted to its own four or five runs, and a two-parameter curve can pass through any two of them. What the plot tests is the shape: whether a step count linear in $1/b$ fits all the runs, with two or three runs per model to spare. The runs of both models scatter around the one curve. Moving along it trades one resource for the other. Toward the upper left, below the critical batch size, a run needs many steps and few examples. Toward the lower right, it needs few steps and many examples. More total work can thus reduce the number of sequential steps and, with enough hardware, the time. Which point is best depends on what is scarce, accelerators or days. This is why large training runs pick batch sizes near their critical batch size, and why the exercises trace this tradeoff for wall-clock time as well as for compute.

9.8.3 Learning-Rate Rules

So far the learning rate moved with the batch by a rule we have not yet justified. Some rule is needed, because a learning rate tuned at one batch size is tuned for the noise of that batch size. Holding η fixed in the sweeps above would have mixed the effect of the batch with that of a mistuned step. Two rules cover practice:

$$\eta(b) = \frac{b}{b_0}\,\eta_0 \quad \text{(SGD)}, \qquad \eta(b) = \sqrt{\frac{b}{b_0}}\,\eta_0 \quad \text{(adaptive methods)}. \tag{9.57}$$

Here η_0 is the learning rate tuned at batch size b_0. The first rule moves the learning rate in proportion to the batch, the second in proportion to its square root.

9.8.3.1 Linear and Square-Root Scaling

The linear rule for SGD (Goyal et al. 2017) follows from how SGD behaves near a minimum. With a constant learning rate, SGD does not settle at the minimum but fluctuates around it in a *noise ball*. For a quadratic with curvature λ and gradient noise of variance σ^2 per example, Section 9.3 found the expected squared distance from the minimum to settle at about

$$\mathbb{E}[x_\infty^2] \approx \frac{\eta\,\sigma^2}{2\lambda b},$$

where the factor $1/b$ is the $1/b$ law for a minibatch of b examples. The noise ball depends on η and b only through η/b. A batch k times larger with a learning rate k times larger therefore keeps the same noise ball and covers the same distance per example, in k times fewer steps. The rule has a limit built in. In the noisy-quadratic model of Equation 9.56, the best learning rate is $\eta_{\text{opt}}(b) = \eta_{\max}/(1 + b_{\text{noise}}/b)$ (McCandlish et al. 2018), where $\eta_{\max}$ is the best step for the exact gradient. It grows linearly with b while $b \ll b_{\text{noise}}$ and saturates beyond. The linear rule is the small-batch half of this curve, and it stops working at the noise scale.

For Adam the rule is weaker. Adam divides each coordinate of its update by the root-mean-square of that coordinate's recent gradients. While the noise dominates, that denominator is about the noise amplitude $\sigma/\sqrt{b}$, so a k times larger batch shrinks it by $\sqrt{k}$ and already makes the signal in each step $\sqrt{k}$ times larger. Covering the same distance per example in k times fewer steps needs a further factor of $\sqrt{k}$, and the square-root rule supplies it. The stochastic-differential-equation analysis of Malladi et al. (2022) makes this precise. Growing the batch by a factor k preserves Adam's dynamics if every time constant moves with it: $\eta' = \sqrt{k}\,\eta$, $1-\beta_i' = k(1-\beta_i)$, and $\epsilon' = \epsilon/\sqrt{k}$. The condition on β_i can be met only while $k(1-\beta_i) < 1$, that is, for $k < 10$ at $\beta_1 = 0.9$. Common practice, and the sweeps of this section, move η alone, which is the form that Equation 9.57 states.

9.8.3.2 Learning-Rate Sweeps at Two Batch Sizes

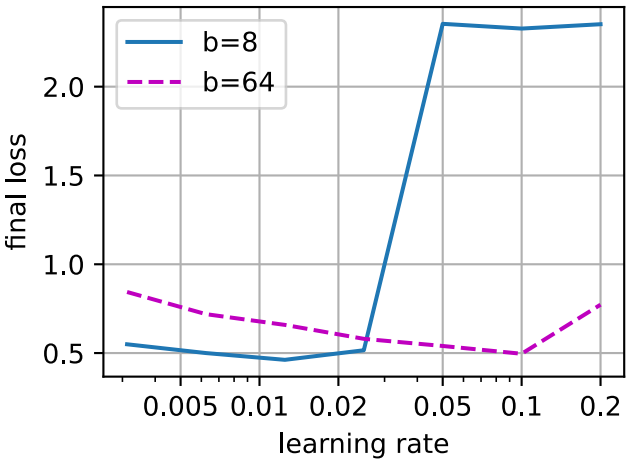
We test both rules on the CNN with learning-rate grids at $b = 8$ and $b = 64$. Every run sees the same budget of 12,800 examples, so the larger batch takes eight times fewer steps. If a rule holds, the whole curve of final loss against η shifts to the right by the predicted factor when the batch grows eightfold: by 8 for SGD and by $\sqrt{8} \approx 2.8$ for Adam. Both batch sizes lie below the critical batch size of two to four hundred images that we fitted for the CNN. There extra examples per step still turn into fewer steps, so both rules should hold. `d2l.lr_sweep` of Section 9.5 runs the grid, quietly, and `d2l.final_loss` summarizes each run, so the sweep takes a few lines. The budget is a fifth of the training set, so every minibatch is new to the model, and its loss, recorded before the update, is a loss on unseen images. We average it over the last quarter of each run, the last 3,200 images at either batch size, which smooths out the jitter of the final weights. Each sweep draws one seed, so all its runs start from the same initialization and see the same images in the same order, and the differences along a curve come from the learning rate alone.

```
def final_loss_at(make_optimizer, lrs, b, seed, num_examples=12800):
    """Train the CNN once per learning rate; return the final losses."""
    curves, _ = d2l.lr_sweep(d2l.FashionCNN, make_optimizer,
                             d2l.FashionMNIST(batch_size=b), lrs, seed,
                             num_examples // b, verbose=False)
    return [d2l.final_loss(c, k=len(c) // 4) for c in curves.values()]
```

First SGD with momentum 0.9, on a grid of learning rates spaced by factors of two:

```
sgd_lrs = [0.003125, 0.00625, 0.0125, 0.025, 0.05, 0.1, 0.2]
make_sgd = lambda model, lr: torch.optim.SGD(model.parameters(), lr,
                                             momentum=0.9)
seed = random.getrandbits(32)
sgd_small = final_loss_at(make_sgd, sgd_lrs, b=8, seed=seed)
sgd_big = final_loss_at(make_sgd, sgd_lrs, b=64, seed=seed)
for b, loss in ((8, sgd_small), (64, sgd_big)):
    print(f'b={b:2d}: best lr {sgd_lrs[np.nanargmin(loss)]:g}')
d2l.plot(sgd_lrs, [sgd_small, sgd_big], 'learning rate', 'final loss',
         xscale='log', legend=['b=8', 'b=64'])
```

```
b= 8: best lr 0.0125
b=64: best lr 0.1
```



For Adam the predicted shift, $\sqrt{8} \approx 2.8$, falls between two grid points, and in a flat basin the best grid point can move by a whole step from one run to the next. We therefore locate each minimum between the grid points: fit a parabola in $\log \eta$ through the lowest final loss and its two neighbors, and take its vertex.

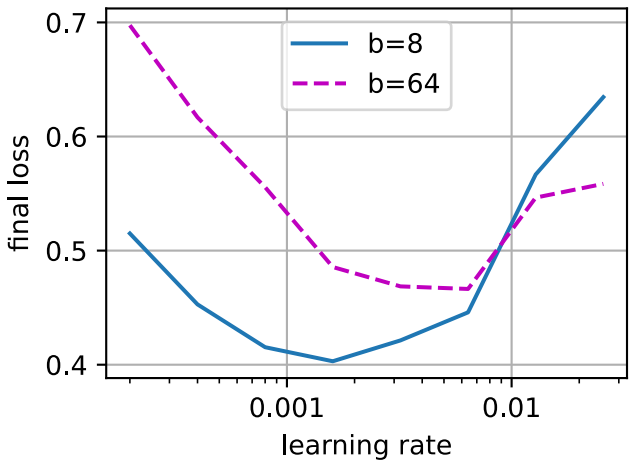
```
def best_lr(lrs, losses):
    """Vertex of a parabola in log(lr) through the minimum and neighbors."""
    i = int(np.clip(np.nanargmin(losses), 1, len(lrs) - 2))
    c2, c1, _ = np.polyfit(np.log(lrs[i - 1:i + 2]), losses[i - 1:i + 2], 2)
    return math.exp(-c1 / (2 * c2))
```

Then Adam:

```
adam_lrs = [0.0002, 0.0004, 0.0008, 0.0016, 0.0032, 0.0064, 0.0128,
            0.0256]
make_adam = lambda model, lr: torch.optim.Adam(model.parameters(), lr)
seed = random.getrandbits(32)
adam_small = final_loss_at(make_adam, adam_lrs, b=8, seed=seed)
adam_big = final_loss_at(make_adam, adam_lrs, b=64, seed=seed)
lr_small, lr_big = best_lr(adam_lrs, adam_small), best_lr(adam_lrs, adam_big)
```

```
print(f'best lr: {lr_small:.2g} at b=8, {lr_big:.2g} at b=64, '
      f'a factor of {lr_big / lr_small:.1f}')
d2l.plot(adam_lrs, [adam_small, adam_big], 'learning rate', 'final loss',
         xscale='log', legend=['b=8', 'b=64'])
```

```
best lr: 0.0015 at b=8, 0.0046 at b=64, a factor of 3.1
```



Both plots show a broad minimum, with the loss rising on the right, where runs become unstable and, in many sweeps, some fail outright. For SGD the larger batch moves the whole basin to the right: the best learning rate moves by a factor of four to sixteen, the batch ratio to within one grid step, so the linear rule carried over from $b = 8$ lands within one grid step of the best learning rate at $b = 64$. For Adam the fitted optimum moves by a factor between about 1.5 and 5 across our runs, and by about 2.5 on average: close to the $\sqrt{8} \approx 2.8$ of the square-root rule, and short of the batch ratio of 8 in every run. Three points of a flat basin fix each optimum only roughly, so a single sweep scatters around the prediction. The basins are broad, and near the optimum a factor of two in η changes the loss little. The rules bridge large ratios of batch sizes, and a short sweep around their prediction does the fine tuning.

9.8.3.3 Limits of the Rules

The rules fail in three places. Past the noise scale, the best learning rate stops growing, because $\eta_{\text{opt}}(b)$ saturates at $\eta_{\max}$, and a learning rate scaled further overshoots. The CNN sweep held its rate fixed above 512 images for this reason, and the exercises reproduce the effect on `TinyLM`.

At the stability ceiling, curvature caps the learning rate. Gradient descent on a function whose curvature is bounded by L is guaranteed to descend only for $\eta < 2/L$ (Section 9.2), and this bound does not depend on the batch. Once $\eta(b)$ reaches it, the optimum stops moving, so the rules operate only in the room below the ceiling.

Early in training, the noise scale is at its smallest and the curvature often at its largest, so a large batch with a scaled-up learning rate can diverge in the first steps. Learning-rate warmup, which Goyal et al. (2017) introduced for this reason, starts from a small η and raises it over the first steps (Section 9.6).

None of this is a universal law. Across many workloads the curves of Equation 9.56 keep their shape, but their critical batch sizes range over orders of magnitude with the architecture, the dataset, and the optimizer (Shallue et al. 2019). Large batches were once also blamed for worse generalization, on the grounds that they converge to sharp minima that test poorly (Keskar et al. 2017). Much of that gap closes when the learning rate and the length of training are retuned for the larger batch (Shallue et al. 2019). The lasting cost of a large batch is data efficiency, which is what this section measured.

9.8.4 Growing the Batch

The noise scale is not a constant of the problem. We measured it growing within 500 steps, by more than an order of magnitude for the language model, and the reason holds at every scale: $\|\nabla f\|$ falls as the easy progress is made, while the disagreement between examples persists. A fixed batch size is therefore a compromise. A batch chosen for the end of training exceeds the critical batch size at the beginning and wastes examples there, while a batch chosen for the beginning leaves parallelism unused later. Large language-model runs resolve this with a *batch-size ramp*. GPT-3, for example, grew its batch from 32 thousand to 3.2 million tokens over the first several billion tokens of training (Brown et al. 2020). Measurements during pretraining find that the critical batch size grows with the amount of data trained on rather than with the model size (H. Zhang et al. 2024), so the ramp can also follow a measurement instead of a fixed schedule. Merrill et al. (2025) measured the critical batch size as training proceeded and doubled the batch each time the measurement overtook it.

A batch ramp interacts with the learning-rate schedule. In the noise ball of SGD above, η and b enter only through η/b, so raising b at a fixed η reduces the noise as much as lowering η at a fixed b. A batch ramp can therefore do part of the work of learning-rate decay (Smith et al. 2018), and it is designed together with the schedule of Section 9.6 rather than tuned as an independent knob. It also interacts with the optimizer. On the pretraining workloads they studied, Shah et al. (2025) find that Muon (Section 9.7) stays data-efficient up to larger batches than AdamW, which moves the critical batch size of Equation 9.56 to the right.

One question remains open. Fewer, larger steps save wall-clock time only if each step is split across many devices that work in parallel. That is data parallelism, the subject of Chapter 13.

9.8.5 Summary

A minibatch gradient is a signal plus noise, and the two have equal power at the gradient-noise scale $b_{\text{noise}} = \operatorname{tr}\Sigma/\|\nabla f\|^2$, which squared gradient norms at two batch sizes suffice to estimate. Below it, doubling the batch roughly halves the steps to a target at constant total work. Above it, the step count levels off at a floor and the extra examples are wasted. Both models of this section followed the predicted hyperbola, with a fitted critical batch size of the same order as their measured noise scales. Moving between batch sizes requires moving the learning rate, in proportion to the batch for SGD and to its square root for Adam. These rules hold below the noise scale and under the stability ceiling, and they need a warmup. The noise scale grows as the loss falls, which is why large runs grow their batch during training.

9.8.6 Exercises

1. code **The noise scale over training.** `noise_scale` of Section 9.8.1.2 measured `TinyLM` at initialization and after 500 steps of Adam. Repeat that run, at batch size 64, for 3,000

steps and measure the noise scale at steps 0, 25, 50, 100, 200, 500, 1,000, 2,000, and 3,000. At each checkpoint also record the held-out loss that `eval_loss` reports on `X_val` and `Y_val`.

1. Plot the noise scale against this loss. Does the growth continue, slow down, or reverse?
2. Under the rule of Section 9.8.4, doubling b whenever b_{noise} overtakes it, and starting from $b = 4$, at which losses would the doublings land? Interpolate $\log b_{\text{noise}}$ linearly in the loss between checkpoints. Explain why the step counts of this run at $b = 64$ would not tell a ramped run when to double.

2. code **Past the elbow.** Extend the `TinyLM` sweep of Section 9.8.2 to $b = 1024$ under the square-root rule of Equation 9.57, then rerun $b = 1024$ with η held at its $b = 256$ value.

1. How do the step counts of the two $b = 1024$ runs compare with each other and with the $b = 256$ run?
2. Explain both observations with Equation 9.56 and the saturating optimal step size $\eta_{\text{opt}}(b) = \eta_{\max}/(1 + b_{\text{noise}}/b)$.

3. code **Retuning instead of scaling.** The CNN sweep recorded in `cnn_bs` and `cnn_steps` moved η by the square-root rule, held fixed above 512 images, rather than retuning it. For each batch size, run a three-point learning-rate grid spaced by factors of two around the rule's value and keep the best steps-to-target. Which points of the curve move? Refit with `fit_steps`: does b_{crit} move, and by more than the difference between two repetitions of the original sweep?

4. **Time-optimal and compute-optimal batch size.** Suppose one optimizer step at batch size b takes $t(b) = t_0(1 + b/b_{\text{sat}})$ seconds, where b_{sat} is the batch size that saturates the device. Time-to-target is then $T(b) = t(b)\,S(b)$, and compute-to-target is the example count $E(b) = b\,S(b)$, which by Equation 9.56 is smallest at the smallest batch. Using Equation 9.56, find the batch size that minimizes $T(b)$, and express its $E(b)$ as a multiple of $E_{\min}$.

5. code **Noise scale per parameter group.** ▯ The parameter groups of Section 9.5.2 split `TinyLM`'s parameters into three groups: embeddings, matrices, and vectors (the biases and the LayerNorm scales and shifts). Restrict `grad_sq_norm` to one group at a time and measure its noise scale with `noise_scale`, at initialization and after 500 steps of Adam.

1. Which group has the largest noise scale, and does the ranking change during training?
2. An embedding row receives a gradient only when its token occurs. Count the rows of `token_emb` that receive no gradient in a probe minibatch of 8 sequences, and state whether this sparsity can account for the embeddings' noise scale. Why does the same count for `pos_emb` always come out zero?
3. What does the ranking imply for one global batch size serving all three groups?

6. code **The factor-of-two check.** ▯ Equation Equation 9.56 predicts that examples-to-target reach twice their minimum at $b = b_{\text{noise}}$, where b_{noise} is averaged over the run.

1. From the checkpoints of problem 1, average b_{noise} over the training steps before the loss first falls to 1.9, interpolating linearly between checkpoints.
2. Compare this average with the b_{crit} that `fit_steps` returns for `lm_bs` and `lm_steps`. By what factor do the two differ?

3. Repeat the sweep and the checkpoint measurements with two more seeds. Is the gap between the two estimates larger than the seed-to-seed scatter of each?

7. [extended] **Measured time-compute tradeoff.** ▯ The step-time model $t(b)$ of problem 4 has two constants, t_0 and b_{sat}, that depend on the hardware.
   1. Measure t_0 and b_{sat} for `TinyLM` on your hardware with `d2l.Timer` by timing steps at several batch sizes.
   2. Combine $t(b)$ with `lm_bs` and `lm_steps` and plot $T(b)$ against $E(b)$. Compare the time-optimal batch size on this curve with the prediction of problem 4, evaluated at the run-averaged b_{noise} of problem 6, and say by what factor the time-optimal batch cuts $T(b)$ relative to $b = 4$ and by what factor it raises $E(b)$.

9.9 Scaling Up

Every experiment in this chapter so far chose its learning rate with a sweep: a few dozen short runs, each lasting seconds to minutes on one GPU. A model with billions of parameters, trained for months on thousands of accelerators, allows no such sweep. Even a single extra full run may be too expensive. Large-scale training therefore tunes a small proxy model and reuses the hyperparameters it finds for the large one. If they do not carry over, the large run diverges or trains slowly, and the error shows only after much of the budget is spent.

This section asks when such a transfer works. We train a family of networks that differ only in width, watch the best learning rate move as the network widens, and trace the move to the size of the very first update. We then change how each layer's learning rate and output scale depend on the width, so that a learning rate tuned on the narrowest network stays good for the widest. A diagnostic that takes seconds checks the change, a sweep tests the transfer, and a last step connects the rules to the update sizes of Muon (Section 9.7).

```
import random
```

9.9.1 Learning Rate and Width

9.9.1.1 A Family of Widths

We want the smallest experiment in which a model can be scaled up while remaining the same model. A multilayer perceptron on Fashion-MNIST serves: three hidden layers of width n, with n from 128 to 1,024. Every member of the family computes the same kind of function; only the width differs. Note where the width lives. The input matrix always has fan-in 784 and the output matrix always has fan-out 10, so only the two $n \times n$ matrices in the middle grow in both dimensions. This distinction matters below. Because we will train several dozen of these models for a few hundred steps each, we keep the whole training set on the accelerator as one tensor. The 60,000 images occupy less than 200 MB, and indexing them directly avoids the overhead of a data loader in such short runs.

```
fashion = d2l.FashionMNIST()
device = d2l.try_gpu()
X = (fashion.train.data.float().reshape(-1, 784) / 255).to(device)
```

```
Y = fashion.train.targets.to(device)
```

We specify the initialization explicitly, because the initialization is part of what a rule for scaling must prescribe: every weight is Gaussian with variance $1/\text{fan-in}$, and every bias is zero. Together with one learning rate shared by all layers, this is the *standard parametrization* (SP). Up to constants it is what default initializers do and what the Xavier and He schemes of Section 4.4 prescribe. The optimizer is created by a method, `configure_adam`, so that a variant of the class can override it later.

```
class MLP(nn.Module):
    """A three-hidden-layer ReLU network under standard parametrization."""
    def __init__(self, width):
        super().__init__()
        self.fc_in = nn.Linear(784, width)
        self.fc_h1 = nn.Linear(width, width)
        self.fc_h2 = nn.Linear(width, width)
        self.fc_out = nn.Linear(width, 10)
        for lin in (self.fc_in, self.fc_h1, self.fc_h2, self.fc_out):
            nn.init.normal_(lin.weight, std=lin.in_features ** -0.5)
            nn.init.zeros_(lin.bias)

    def features(self, X):
        h = F.relu(self.fc_h1(F.relu(self.fc_in(X))))
        return F.relu(self.fc_h2(h))

    def forward(self, X):
        return self.fc_out(self.features(X))

    def configure_adam(self, lr):
        return torch.optim.Adam(self.parameters(), lr)
```

Training is a step-counted loop in the style of Section 9.5: 400 steps of Adam at batch size 512, about three passes over the data, at a constant learning rate. A seed fixes both the initialization and the sequence of minibatches, so runs with the same seed differ only in width, parametrization, and learning rate. The score of a run is its cross-entropy on the whole training set after the last step. A diverged run is capped at 2.5, slightly above the loss $\log 10 \approx 2.3$ of uniform guessing.

```
def train_step(model, optimizer, Xb, Yb):
    loss = F.cross_entropy(model(Xb), Yb)
    optimizer.zero_grad()
    loss.backward()
    optimizer.step()
    return loss.detach()

def train_mlp(arch, width, lr, seed, num_steps=400, batch_size=512):
    torch.manual_seed(seed)
    model = arch(width).to(device)
    optimizer = model.configure_adam(lr)
    idx = np.random.default_rng(seed).integers(0, X.shape[0],
```

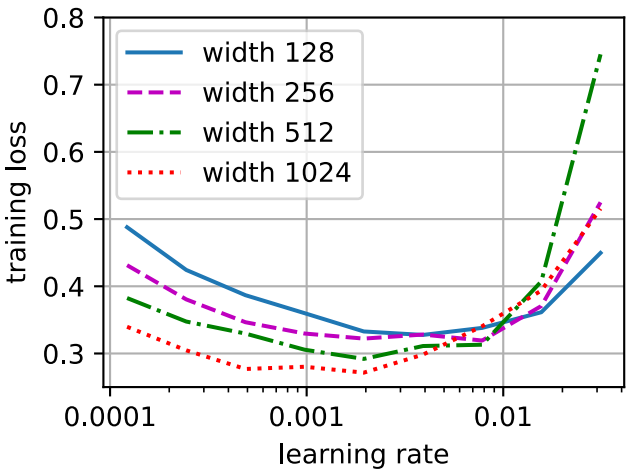
```
                                          (num_steps, batch_size))
    for i in idx:
        train_step(model, optimizer, X[i], Y[i])
    with torch.inference_mode():
        v = float(F.cross_entropy(model(X), Y))
    return min(v, 2.5) if math.isfinite(v) else 2.5
```

9.9.1.2 Learning-Rate Sweep

We evaluate four widths and nine learning rates spaced by factors of two, one short run per configuration, all from one random seed. The 36 runs stand in for the proxy sweep of a large project. They show whether the learning rate selected at one width serves the others.

```
widths = [128, 256, 512, 1024]
lrs = [2 ** k for k in range(-13, -4)]
seed = random.getrandbits(32)  # One draw for every run of this section
sp_loss = {w: [train_mlp(MLP, w, lr, seed) for lr in lrs] for w in widths}
for w in widths:
    print(f'width {w:4d}: best lr {min(zip(sp_loss[w], lrs))[1]:.1e}')
d2l.plot(lrs, [sp_loss[w] for w in widths], 'learning rate',
         'training loss', xscale='log', ylim=[0.25, 0.8],
         legend=[f'width {w}' for w in widths])
```

```
width  128: best lr 3.9e-03
width  256: best lr 7.8e-03
width  512: best lr 2.0e-03
width 1024: best lr 2.0e-03
```



Each width traces a U: too small a learning rate undertrains, and too large a one destabilizes. On the right the wider networks tend to climb fastest, in some runs off the top of the chart, where the capped runs sit. The U's do not line up. The minimum slides left as the network widens: the best learning rate at width 1,024 is several times smaller than at width 128, and the drift continues at larger widths (Yang et al. 2022). After tuning, the wider networks match or beat the narrower ones, so width helps. But the best region does not transfer. The

width-128 optimum puts the width-1,024 network on the unstable side of its U, and the width-1,024 optimum undertrains the narrow network. Under standard parametrization the right learning rate depends on the width as well as on the task.

9.9.1.3 The First Adam Step

The size of the first update explains the drift and shows how to correct it. Consider a hidden weight matrix $\mathbf{W} \in \mathbb{R}^{n \times n}$ that receives activations $\mathbf{h}$ whose entries have typical size $\overline{|h|}$. For a single example its gradient is an outer product, $\mathbf{g} = \delta \mathbf{h}^\top$, where δ is the error propagated back to the layer's output. On the very first step, Adam's ratio $\hat{\mathbf{m}}/\sqrt{\hat{\mathbf{v}}}$ equals $\operatorname{sign}(\mathbf{g})$, because after bias correction the first and second moment estimates are $\mathbf{g}$ and $\mathbf{g}^2$ (Section 9.5). The sign of an outer product factorizes, $\operatorname{sign}(\delta_i h_j) = \operatorname{sign}(\delta_i)\operatorname{sign}(h_j)$, so the update changes the layer's output by

$$(\Delta \mathbf{W}\,\mathbf{h})_i = -\eta \sum_{j=1}^{n} \operatorname{sign}(\delta_i h_j)\, h_j = -\eta\, \operatorname{sign}(\delta_i) \sum_{j=1}^{n} |h_j| \approx -\eta\, \operatorname{sign}(\delta_i)\, n\, \overline{|h|}. \tag{9.58}$$

All n terms of the sum have the same sign, because the update is built from the very activations that it multiplies. Their sizes add up instead of cancelling, and each output coordinate moves by about $\eta\, n\, \overline{|h|}$. A minibatch gradient is a sum of such outer products, and its sign no longer factorizes, so this argument holds only to leading order. The coordinate check below confirms the width dependence by measurement. Doubling the width doubles the change, so the largest stable learning rate falls in proportion to $1/n$, the trend that the sweep shows. The effect differs between layers. The input layer has fan-in 784 at every width, and a bias has no fan-in at all, so their stable learning rates do not move. The output matrix has fan-in n as well, so its step grows like those of the hidden matrices. It gets its own correction below. A single global η is a compromise between layers that scale differently, and the layers with fan-in n pull it down.

9.9.2 Maximal Update Parametrization

9.9.2.1 The Rules

If the stable learning rate of each layer scales differently with width, one learning rate should not have to serve them all. The *maximal update parametrization* (muP) (Yang et al. 2022) writes the width dependence into per-layer rules. Relative to a chosen *base width*, each layer's learning rate, and one scale factor in the forward pass, absorbs that layer's own scaling. What remains is a base learning rate that does not depend on the width and can be tuned on the smallest member of the family. The rules are derived from the limit of infinite width in which every layer's activations stay of order one *and* every layer keeps learning features (Yang and Hu 2021). Larger updates make activations grow with width, as in Equation 9.58; smaller ones freeze the features in the limit. "Maximal update" names the largest updates that keep the activations stable.

For Adam and a width multiplier $m = n/n_{\text{base}}$, the rules fit in a table.

:The maximal update parametrization for Adam, relative to a chosen base width. The hidden-matrix learning rate and the output-logit multiplier depend on the width; the initialization, the learning rates of the input weights and biases, and the rest of the forward pass do not.

Table 9.2: Mup Rules

parameters	initialization	Adam learning rate	forward pass
input weights and all biases	unchanged	η	unchanged
hidden matrices	variance $\propto 1/\text{fan-in}$ (unchanged)	η/m	unchanged
output matrix	unchanged	η	logits $\times\, 1/m$

The initialization column is the one that does *not* change: variance $\propto 1/\text{fan-in}$ is right at every width, and `MLP` already uses it. The hidden learning rate shrinks as $1/m$, which cancels the n-fold sum of Equation 9.58. The output layer keeps its learning rate, but its logits are divided by m, which cancels the same sum on the output side while letting the head keep learning. Input weights and biases, whose fan-in never grows, are left alone. For language models the token embedding belongs to this class too, which is why muP treats embeddings apart from hidden matrices. Transformers need one further rule: attention logits scaled by $1/d$ rather than $1/\sqrt{d}$. SGD has its own column of exponents, and several algebraically equivalent formulations are in use. The practitioner's guide of Dey et al. (2024) lays out the full table and its pitfalls. Here the whole change is a subclass:

```
class MuMLP(MLP):
    """The same network under muP, relative to a width-1,024 base."""
    def __init__(self, width, base_width=1024):
        super().__init__(width)
        self.m = width / base_width

    def forward(self, X):
        # Rule 1: scale the output matrix's logits; the bias is untouched
        return self.fc_out(self.features(X) / self.m)

    def configure_adam(self, lr):
        hidden = [self.fc_h1.weight, self.fc_h2.weight]
        rest = [p for p in self.parameters()
                if not any(p is q for q in hidden)]
        return torch.optim.Adam([              # Rule 2: hidden LR / m
            {'params': rest, 'lr': lr},
            {'params': hidden, 'lr': lr / self.m}])
```

At the base width $m = 1$, and `MuMLP` is `MLP`. Which width serves as the base is a free choice. Every choice gives the same dependence on the width; choices differ only by constant factors per layer, such as how fast the input layer learns relative to the hidden ones and how large the output multiplier is. These constants are hyperparameters in their own right, and the practitioner's guide tunes them on the proxy along with the learning rate. We avoid those extra sweeps by putting the base at the widest member of the family, 1,024, where `MuMLP` is the standard network with one learning rate for all layers. The narrower members, including the width-128 model that we tune, have $m < 1$. A base at width 128 would instead fix the constants where they matter least: in our runs it leaves tuned muP at width 1,024 one to three hundredths of a nat above tuned standard parametrization. An exercise traces which constants make the difference.

9.9.2.2 The Coordinate Check

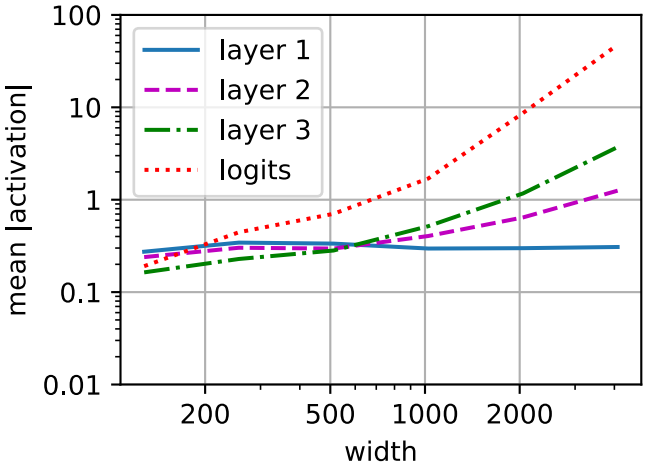
The *coordinate check* tests the width scaling of Equation 9.58 directly, and the practitioner's guide of Dey et al. (2024) recommends it as the first test of any muP implementation. Under the rules used here, the hidden activations should stay of order one after a training step at every width, while the logits follow their explicit $1/m$ scaling. Activations that grow with width reveal a scaling error. We therefore build the family across widths, apply one Adam update at a learning rate in the middle of the grid, and record the mean absolute activation of each layer. One step is the cleanest probe: Equation 9.58 describes it directly, while later steps compound the growth and mix it with feedback from the loss.

```
def activations(model, Xb):
    with torch.inference_mode():
        h1 = F.relu(model.fc_in(Xb))
        h2 = F.relu(model.fc_h1(h1))
        h3 = F.relu(model.fc_h2(h2))
        return [float(h.abs().mean()) for h in (h1, h2, h3, model(Xb))]

def coord_check(arch, widths, seed, lr=2**-8, num_steps=1):
    Xb, Yb = X[:256], Y[:256]
    sizes = []
    for width in widths:
        torch.manual_seed(seed)
        model = arch(width).to(device)
        optimizer = model.configure_adam(lr)
        for _ in range(num_steps):
            train_step(model, optimizer, Xb, Yb)
        sizes.append(activations(model, Xb))
    return list(zip(*sizes))
```

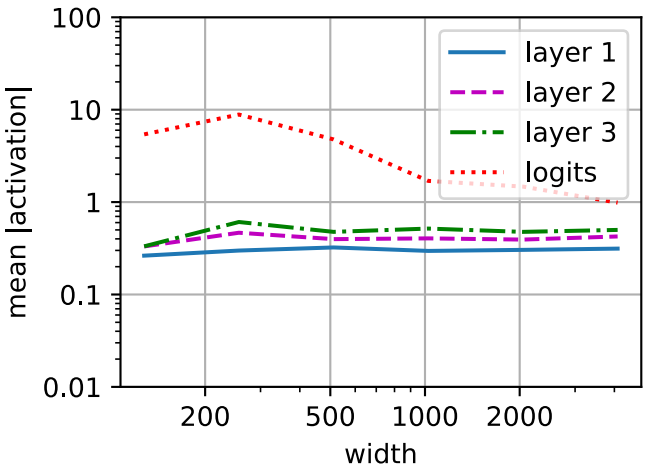
First under standard parametrization. The check takes seconds, so we extend the widths to 4,096, beyond those of the training sweep:

```
check_widths = [128, 256, 512, 1024, 2048, 4096]
sp_acts = coord_check(MLP, check_widths, seed)
d2l.plot(check_widths, list(sp_acts), 'width', 'mean |activation|',
         xscale='log', yscale='log', ylim=[0.01, 100],
         legend=['layer 1', 'layer 2', 'layer 3', 'logits'])
```



The plot shows Equation 9.58 at work. Layer 1, behind the input matrix with its fixed fan-in, is flat. Layers 2 and 3, behind the square matrices, grow with width, and the growth compounds with depth because each layer passes its excess on to the next. The logits grow by about two orders of magnitude across the range. A first step that is stable at narrow widths therefore destabilizes a wide enough model. The same check under muP, on the same axes:

```
mup_acts = coord_check(MuMLP, check_widths, seed)
d2l.plot(check_widths, list(mup_acts), 'width', 'mean |activation|',
         xscale='log', yscale='log', ylim=[0.01, 100],
         legend=['layer 1', 'layer 2', 'layer 3', 'logits'])
```



All three layer curves are flat: no layer's update size depends on the width anymore. The logits *fall* with width, and this is by design. The $1/m$ multiplier keeps the logits of the wider networks small at first, and they grow to order one through learning rather than through size. Growth is the unambiguous failure. A flat or falling curve says only that the forward pass is stable, because the check reads the sizes of activations, not whether features are

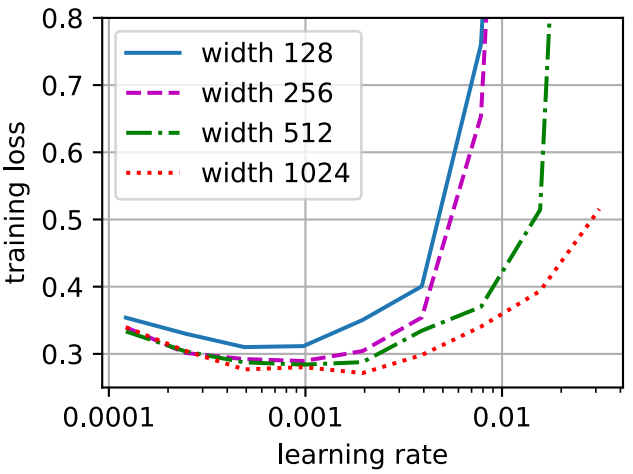
learned. Updates that shrink too fast with width would also produce falling curves, with the layer frozen. The test still catches the common implementation errors: a missing multiplier, a mislabeled layer, or an initializer with a fixed standard deviation, like the 0.02 common in transformer code, in place of one that scales with the fan-in. The exercises break each of the two rules in turn and ask which errors the check detects.

9.9.2.3 Learning-Rate Transfer

The coordinate check tests the scaling of activations. Whether the learning rate now transfers is a question for the sweep, which we repeat under muP with the same seed:

```
mup_loss = {w: [train_mlp(MuMLP, w, lr, seed) for lr in lrs] for w in widths}
for w in widths:
    print(f'width {w:4d}: best lr {min(zip(mup_loss[w], lrs))[1]:.1e}')
d2l.plot(lrs, [mup_loss[w] for w in widths], 'learning rate',
         'training loss', xscale='log', ylim=[0.25, 0.8],
         legend=[f'width {w}' for w in widths])
```

```
width  128: best lr 4.9e-04
width  256: best lr 9.8e-04
width  512: best lr 9.8e-04
width 1024: best lr 2.0e-03
```



The width-1,024 curve repeats the earlier one, because at the base width the two parametrizations coincide and the seed is the same. The narrower curves have moved to meet it, and the minimum no longer drifts with width: across the eightfold change in width, each grid minimum lies within one or two grid steps of the width-128 minimum. To measure what transfer is worth, we reuse the learning rate selected at width 128 for width 1,024 and compare the result with the best loss of the width-1,024 sweep. We also print the best loss at each width, the result of tuning every width separately:

```
for name, loss in (('SP', sp_loss), ('muP', mup_loss)):
    tuned = loss[128].index(min(loss[128]))  # Grid index chosen at width 128
```

```
    excess = loss[1024][tuned] / min(loss[1024]) - 1
    best = '  '.join(f'{min(loss[w]):.3f}' for w in widths)
    print(f'{name:3s}  best loss at widths {widths}: {best}')
    print(f'     width-128 lr at width 1024: {excess:.0%} above its best')
```

```
SP   best loss at widths [128, 256, 512, 1024]: 0.328  0.319  0.292  0.271
     width-128 lr at width 1024: 10% above its best
muP  best loss at widths [128, 256, 512, 1024]: 0.310  0.289  0.284  0.271
     width-128 lr at width 1024: 2% above its best
```

Under muP the transferred learning rate typically lands within a few percent of the best loss at width 1,024. Under standard parametrization it typically lands 10% to 30% above it. Each number comes from a single seed and varies from run to run, so an occasional run brings the two close together; the first exercise measures the spread. This is *hyperparameter transfer*: tune the small model, scale up, keep the numbers. Under muP the best loss also falls from width 128 to width 1,024. Retuned at width 1,024, standard parametrization reaches the same best loss, because there the two parametrizations coincide. On a family this small that sweep is still cheap; muP saves it. Yang et al. (2022) find that the two diverge at larger widths: the tuned loss under standard parametrization stops improving, while under muP it keeps falling. Transfer concerns the base learning rate, not every hyperparameter: schedules, batch size (Section 9.8), and regularization still interact with scale. At a large scale, Yang et al. (2022) tuned a 6.7-billion-parameter GPT-3 by sweeping a 40-million-parameter proxy, spent about 7% of the pretraining compute on tuning, and outperformed the original model.

9.9.3 The Spectral View

muP arrived through limits of infinite width, but a shorter route connects it to Section 9.7. There we measured the size of a matrix update by its spectral norm, the largest factor by which it can stretch an activation vector. For a layer that maps n_{in} activations to n_{out}, with entries of order one on both sides, the healthy size is a spectral norm of order $\sqrt{n_{\text{out}}/n_{\text{in}}}$, for the weights and for their updates alike:

$$\|\mathbf{W}\|_2 \asymp \sqrt{\frac{n_{\text{out}}}{n_{\text{in}}}}, \qquad \|\Delta\mathbf{W}\|_2 \asymp \sqrt{\frac{n_{\text{out}}}{n_{\text{in}}}}. \tag{9.59}$$

An input with entries of order one has length $\sqrt{n_{\text{in}}}$, and an output with entries of order one has length $\sqrt{n_{\text{out}}}$, so a matrix that maps one to the other stretches by the ratio of the two. Yang et al. (2023) show that this *spectral condition* recovers muP. Adam's first update of a hidden $n \times n$ matrix, the rank-one sign matrix behind Equation 9.58, has a spectral norm of order ηn, about $\eta n/2$ when half the ReLU units are active (an exercise derives it). The condition asks for order one, and dividing the hidden learning rate by m removes the excess factor n/n_{base}. Seen this way, the per-layer rules of Table 9.2 bring Adam's coordinatewise updates to the spectral scale at every width.

An optimizer whose update is already a matrix of controlled spectral norm needs fewer of these corrections. Muon replaces the update of each hidden matrix by an orthogonalized one, whose singular values all equal one (Jordan et al. 2024; Bernstein and Newhouse 2024). Multiplied by $\sqrt{n_{\text{out}}/n_{\text{in}}}$ (Bernstein 2025), that update meets Equation 9.59 at every width by construction. The scale used in Section 9.7, $0.2\sqrt{\max(n_{\text{out}}, n_{\text{in}})}$, was chosen instead to

match the root-mean-square entry of an AdamW update. It gives a square $n \times n$ update a spectral norm of $0.2\,\eta\sqrt{n}$, which still grows with width, though more slowly than the ηn of Adam's first step. Muon at that scale therefore still needs width corrections for its learning rate to transfer, and Shah et al. (2025) pair it with muP to transfer hyperparameters across model sizes.

Width is only one direction of scaling. Transfer across depth, batch size, and training length needs rules of its own, and whether weight decay rather than parametrization keeps long runs stable is still debated (Kosson et al. 2025). The coordinate check is worth running on any family of models one intends to scale, whatever the parametrization.

9.9.4 Summary

Hyperparameters tuned on a small model do not transfer by default: the best learning rate falls as the network widens, because one Adam step perturbs a hidden layer's activations in proportion to its fan-in (Equation 9.58). The maximal update parametrization removes the width dependence with per-layer rules: the usual initialization with variance $\propto 1/\text{fan-in}$, hidden-matrix learning rates divided by the width multiplier m, logits divided by m, and input weights and biases left alone. These rules fix how each layer scales with the width; the choice of base width fixes the constant factors. In our family of networks, with the base at the widest member, the best learning rate then stays within a grid step or two of its width-128 value across an eightfold change in width. The coordinate check tests such rules in seconds: after a training step, activations should follow the width dependence that the parametrization prescribes, without unintended growth. In spectral terms, muP makes Adam's updates land at the norm that a layer's shape prescribes, and Muon's orthogonalized updates carry part of that scaling by construction.

9.9.5 Exercises

1. code **Transfer across seeds.** `sp_loss` and `mup_loss` hold one run per width and learning rate, all from one seed, so the transfer cost that they show carries no error bar.
    1. Under each parametrization, sweep `lrs` at widths 128 and 1,024 for three seeds. For each seed, apply that seed's width-128 optimum at width 1,024 and report, in percent, how far its loss lies above the minimum of the same seed's width-1,024 sweep.
    2. Is the difference between the two parametrizations' transfer costs larger than the spread of each cost across seeds?
2. code **Breaking muP on purpose.** The coordinate check is a test for implementation errors.
    1. Delete the $1/m$ logit scaling from `MuMLP` but keep the hidden learning-rate rule and rerun `coord_check`. Which curve exposes the bug?
    2. Restore the multiplier and instead give the hidden matrices the full learning rate. Which curve exposes this bug?
    3. Restore the hidden learning-rate rule, but divide the learning rate by m^2 instead of m. Does `coord_check` flag this error? Sweep `lrs` at width 1,024 with this variant, and compare its best learning rate and best loss with those of `MuMLP` in `mup_loss`.
3. code **Choosing the base.** `MuMLP` puts its base at width 1,024, the widest member of the family.
    1. Rerun the muP sweep with `base_width=128`, so that muP coincides with standard parametrization at the tuned width instead. Does the best learning rate still trans-

fer from width 128 to width 1,024? Compare the best loss at each width with `mup_loss` and `sp_loss`.
   2. The two bases differ only by constant factors per layer. List them, and find which of them accounts for the difference at width 1,024 by changing one at a time.
4. code **Fitting the drift.** Instead of removing the drift, a large project can measure it: DeepSeek-AI, Bi, et al. (2024) fitted power laws for the optimal learning rate and batch size as functions of the compute budget across many small runs and extrapolated them to the target scale. In the first-step analysis of Equation 9.58, the largest stable learning rate scales as $1/n$.
   1. From `sp_loss`, fit a power law $\eta^\star(n) = c\, n^{-\gamma}$ to the best learning rate at each width. Since `lrs` is spaced by factors of two, each optimum is resolved only to within a factor of $\sqrt{2}$. Give the range of γ consistent with this resolution, and state whether it contains $\gamma = 1$.
   2. Extrapolate the fit to widths 2,048 and 4,096. Sweep directly at those widths with `train_mlp`, extending `lrs` downward if a prediction falls below its smallest value, and report how many grid steps separate each prediction from the measured optimum.
5. code **Depth and the coordinate check.** ▣ `coord_check` varies width at a fixed depth of three hidden layers. Generalize `MLP` and `MuMLP` to L hidden layers and run the check across $L \in \{3, 6, 12, 24\}$ at width 512 under both parametrizations. The variance-$1/$fan-in initialization of `MLP` halves the second moment of the ReLU activations at every layer (Section 4.4.2.2), so for this problem initialize the input and hidden matrices with variance $2/$fan-in.
   1. Under standard parametrization, how does the growth of the logits with depth compare with their growth with width in Section 9.9.2.2?
   2. Does muP's width scaling keep the activations flat across depth? State what the rules of Table 9.2 promise and what they do not.
6. **First update in spectral norm.** ▣ For a single example, Adam's first update of a weight matrix $\mathbf{W} \in \mathbb{R}^{n_{\text{out}} \times n_{\text{in}}}$ is $\Delta\mathbf{W} = -\eta\ \text{sign}(\delta)\ \text{sign}(\mathbf{h})^\top$, as in the derivation of Equation 9.58. Let k_{out} and k_{in} be the numbers of nonzero entries of δ and $\mathbf{h}$.
   1. Compute $\|\Delta\mathbf{W}\|_2$ as a function of η, k_{out}, and k_{in}.
   2. Assume that k_{out} and k_{in} are fixed fractions of n_{out} and n_{in}. For the input, hidden, and output matrices of `MLP`, find how η must scale with the width n for $\Delta\mathbf{W}$ to satisfy the spectral condition Equation 9.59.
   3. Show that the learning rates and the logit multiplier of Table 9.2 give every layer an update at this scale.
7. code **Weight decay and parametrization.** ▣ Kosson et al. (2025) argue that in long runs weight decay, not parametrization, stabilizes the update dynamics across widths. Override `configure_adam` in a subclass of `MLP` to return AdamW (Section 9.5.4) with decay coefficient $\lambda = 10^{-3}/\eta$, so that every run averages its updates over the horizon $1/(\eta\lambda) = 1{,}000$ steps described in Section 9.5.4.4.
   1. Run the standard-parametrization sweep of Section 9.9.1.2 at widths 128 and 1,024 with `num_steps=4000`, once with this AdamW and once with Adam. Report the best learning rate at each width for both optimizers. Does the optimum drift less with width under AdamW?
   2. `MLP` has no normalization layer. Given the equilibrium argument of Section 9.5.4.4, state what your result can and cannot say about the cited claim.

9.10 Practice

Every experiment in this chapter so far ran in seconds or minutes. We could sweep a learning rate over four values, repeat a comparison, and restart a run that went wrong. A large training run allows none of this. A sweep over four learning rates costs four times the run itself, so most hyperparameters are set once and never tuned on the model that matters. A single oversized update can undo days of progress. PaLM, a language model with 540 billion parameters trained on 6,144 TPU chips, saw its loss spike about twenty times along the way, although gradient clipping was enabled (Chowdhery et al. 2022). And the checkpoint at the end is one draw from a noisy trajectory, yet it is the one that ships. This section asks how to set up, protect and finish a run that can be done only once.

We follow such a run through its life, using the methods of the earlier sections on models small enough to train here. We first ask how a configuration is chosen when the target cannot be tuned, and read the settings that four recent reports disclose. We then protect the run against rare oversized updates with gradient clipping, and ask what to do when a spike gets through anyway. Next we reduce the noise in the final weights by averaging them. Finally we turn to the question behind every comparison in this chapter: what does it take to show that one optimizer is better than another?

```
import copy
```

9.10.1 Choosing a Configuration

A run too large to tune directly takes its hyperparameters from two sources. The first is smaller proxies. Section 9.9 showed that under a suitable parametrization the best learning rate of a narrow model stays the best as the width grows, so most of the sweeping can happen on models that cost a fraction of the target. The second is precedent: the settings that worked for earlier runs of the same kind.

9.10.1.1 Reported Configurations

Some of that precedent is public. Table 9.3 collects the optimizer configurations disclosed by four prominent pretraining reports from 2024 to 2026. Each entry is what a team reported shipping, usually without an ablation, and a dash means that the report does not say.

:What four pretraining reports disclose about their optimizer configuration. A dash means the report does not say. Batch sizes are quoted the way each report quotes them (tokens or sequences).

Table 9.3: Practice Recipes

run	optimizer	β_1, β_2	peak LR and schedule	warmup	clip	weight decay	batch size
Llama 3 405B (Grattafiori et al. 2024)	AdamW	—	8×10^{-5}, cosine to 8×10^{-7}	8k steps	—	—	ramp 4M → 8M → 16M tokens
DeepSeek-V3 (DeepSeek-AI, Liu, et al. 2024)	AdamW	0.9, 0.95	2.2×10^{-4}, constant to 10T tokens, cosine to 2.2×10^{-5}	2k steps	1.0	0.1	ramp 3,072 → 15,360 seqs over 469B tokens
OLMo 2 7B (Team OLMo et al. 2025)	AdamW	0.9, 0.95	3×10^{-4}, cosine to 10% of peak	2k steps	1.0	0.1, embeddings exempt	1,024 seqs, fixed
Kimi K2 (Kimi Team 2025a)	MuonClip	—	2×10^{-4}, constant to 10T tokens, cosine to 2×10^{-5}	500 steps	—	0.1	67M tokens, fixed

Three of the four runs use AdamW. The fourth, Kimi K2, applies Muon to its hidden weight matrices and AdamW to everything else, the split of Section 9.7. The two reports that disclose the moment coefficients both choose $(\beta_1, \beta_2) = (0.9, 0.95)$. With $\beta_2 = 0.95$, Adam's average of the squared gradients reaches back about $1/(1-\beta_2) = 20$ steps instead of the thousand steps of the default $\beta_2 = 0.999$, so it adapts quickly when the scale of the gradients changes. Three of the four decay the weights with $\lambda = 0.1$, and the two that report clipping clip the global gradient norm at 1. All four warm up and later decay the learning rate, either along a cosine from the start or after a long phase at a constant rate, the warmup–stable–decay shape of Section 9.6. Two ramp the batch size up during training, and two keep it fixed. Every report sets its own peak learning rate, and the four values span almost a factor of four, from 8×10^{-5} for the dense 405-billion-parameter model to 3×10^{-4} for the 7-billion one. The dashes record missing evidence, not implied defaults, and even the complete rows fall short of what a reproduction would need.

9.10.1.2 Tuning Budgets

The settings that recur in the table are a defensible starting point for a new run of the same kind, because each was chosen by a team that could test it at scale. They are not proven

optima, because the reports rarely ablate them. How far to depart from them depends on how many runs the budget allows.

With a handful of runs, keep the recurring settings and use every run to tune the peak learning rate. It is the setting that each report chose for itself, and in the sweeps of this chapter a factor of three in the learning rate separated the best run from a diverging one. With tens of runs, add the weight decay and the schedule, and tune jointly only what is coupled. In AdamW each step multiplies the weights by $1 - \eta\lambda$ (Section 9.5.4), so the learning rate η and the weight decay λ set the shrinkage of the weights through their product. A joint grid should therefore vary the product $\eta\lambda$ and also move along lines on which the product is constant. With more runs than that, the tuning becomes a study of its own, and the protocol of Section 9.10.4 applies.

9.10.2 Gradient Clipping

The clipping column of Table 9.3 guards against a failure that this chapter has already met. In Section 9.5, SGD with momentum trained a small transformer best at the learning rate $\eta = 0.3$ and diverged at $\eta = 1$, one point higher on the grid. Global-norm clipping, introduced in Section 8.5 to tame exploding RNN gradients, treats the gradients of all parameters as one long vector $\mathbf{g}$ and shrinks this vector whenever its length exceeds a threshold θ:

$$\mathbf{g} \leftarrow \min\left(1, \frac{\theta}{\|\mathbf{g}\|_2}\right)\mathbf{g}. \tag{9.60}$$

A gradient shorter than θ passes unchanged, and a longer one keeps its direction and is scaled to length θ. At $\theta = 1$, a gradient of norm 5 is multiplied by 0.2. Because the norm is taken over all parameters at once, clipping preserves the relative sizes of the gradients of different layers and shrinks only the step as a whole.

`clip_grad_norm_` rescales the gradients in place and returns the norm it measured before clipping. The wrapper below calls it between `backward` and `step`. It keeps the returned norms on the device, so that recording them adds no synchronization with the host, and we read them once after training.

```
class Clipped:
    """Clip the global gradient norm before every optimizer step."""
    def __init__(self, optimizer, params, max_norm=1.0):
        self.optimizer, self.params = optimizer, list(params)
        self.max_norm, self.norms = max_norm, []

    def step(self):
        self.norms.append(nn.utils.clip_grad_norm_(self.params, self.max_norm))
        self.optimizer.step()

    def zero_grad(self):
        self.optimizer.zero_grad()
```

9.10.2.1 Preventing a Numerical Failure

We now rerun SGD at the divergent point of that sweep, $\eta = 1$, with and without clipping. The model is `TinyLM`, the decoder-only transformer of Section 9.5.2: two blocks, about 0.47

million parameters, trained to predict the next byte of WikiText-103, a corpus of Wikipedia articles, from the 64 bytes before it. The library function `d2l.train_steps` trains it for a fixed number of steps on minibatches of 64 sequences and returns the training loss of every step. Both runs start from the same initial weights and use SGD with momentum 0.9 at $\eta = 1$ for 2,000 steps; they differ only in the clipping. To see how often and when clipping intervenes, we also record the global gradient norm before clipping at every step.

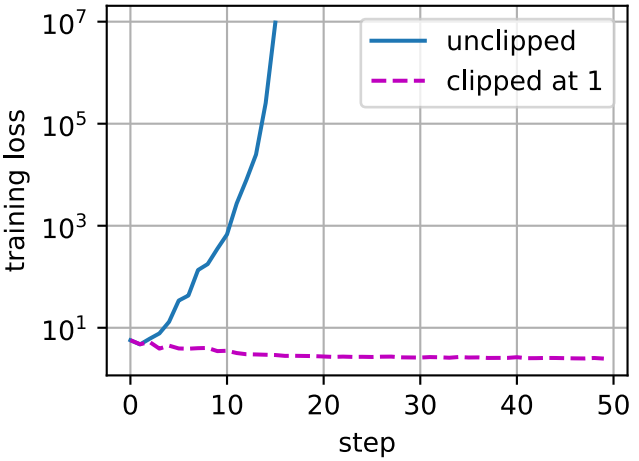
```
data = d2l.WikiText103(batch_size=64, num_steps=64)
```

```
num_steps = 2000
model0 = d2l.TinyLM(len(data.vocab))
curves = {}
for clip in (False, True):
    model = copy.deepcopy(model0)  # Both runs start from the same weights
    optimizer = torch.optim.SGD(model.parameters(), lr=1.0, momentum=0.9)
    if clip:
        optimizer = Clipped(optimizer, model.parameters())
    label = 'clipped at 1' if clip else 'unclipped'
    curves[label] = d2l.train_steps(model, data, optimizer, num_steps)
    print(f'{label}: final loss {sum(curves[label][-100:]) / 100:.3f}')
norms = torch.stack(optimizer.norms)
fired = torch.nonzero(norms > 1).flatten()  # Steps on which clipping acted
print(f'clipped {len(fired)} of {len(norms)} steps, the last at step '
      f'{int(fired.max())}; median gradient norm {norms.median():.2f}')
```

```
unclipped: final loss nan
clipped at 1: final loss 1.614
clipped 11 of 2000 steps, the last at step 10; median gradient norm 0.26
```

The unclipped run ends with a loss of `nan`, “not a number”, the value that floating-point arithmetic produces once a computation has overflowed. The instability occurs early, so we plot the first 50 steps on a log scale.

```
d2l.plot(list(range(50)), [curves[k][:50] for k in curves], 'step',
         'training loss', legend=list(curves), yscale='log')
```



The first step lowers the loss, and from the second on the unclipped run feeds on itself. An oversized step moves the parameters to where the loss is steeper, the gradient there is larger, and momentum carries the excess into the following steps. The loss grows by several orders of magnitude and overflows within about twenty steps. The clipped run, at the same learning rate, trains normally, to a final loss comparable with that of the best SGD run in Section 9.5, which used a learning rate about three times smaller.

9.10.2.2 Clipping Frequency

The printed counts show how little clipping did to achieve this, and when it did it. The median gradient norm of the clipped run is about a quarter of the threshold, and clipping changed the update on a few dozen at most of the 2,000 steps, all of them within the first forty. These are the steps on which the freshly initialized network has its largest gradients. Once they are past, the clipped run trains at $\eta = 1$ with clipping idle for the rest of the 2,000 steps. Here clipping did the job of a warmup, and an adaptive one. It shortened exactly those first steps that were too long, until the parameters reached a region where the full step is safe, and it needed no schedule to tell it when to stop. Every run in Table 9.3 warms up as well, and the exercises compare the two remedies.

Used this way, clipping is a *guard*: it leaves nearly every update alone and rescales only the rare oversized ones. In long runs the same guard has a second job. Gradient noise in language models is heavy-tailed, so long after the start an occasional minibatch produces a gradient many times longer than typical ones, and under noise of this kind clipped SGD provably converges where plain SGD can fail (Zhang et al. 2020).

A guard works only if the threshold lies above the gradient norms of a stable run, so that it fires rarely. If clipping fires on most steps, it rescales nearly every update and thereby changes the update rule itself. The learning rate and the threshold then act together on the length of the step, and they have to be tuned together (Godbole et al. 2023).

9.10.2.3 Clipping and Adam

Clipping protects Adam in a different way. Adam keeps two leaky averages (Section 9.5): $\mathbf{m}$ of the gradients, with weight $1 - \beta_1$ on each new gradient, and $\mathbf{v}$ of their squares, with weight $1 - \beta_2$. Suppose a gradient coordinate g arrives that dwarfs every earlier one. After

this step $m \approx (1-\beta_1)g$ and $v \approx (1-\beta_2)g^2$, so the update of that coordinate is about $\eta(1-\beta_1)/\sqrt{1-\beta_2}$, or $3.2\,\eta$ at the defaults $\beta_1 = 0.9$ and $\beta_2 = 0.999$, however large g is. Adam's update is thus already bounded in each coordinate, and clipping, which scales g before either average sees it, changes this one step little as long as the clipped gradient still dominates. What clipping protects is Adam's state. Without it, the huge g^2 stays in v and fades only by the factor β_2 per step, with a time constant of $1/(1-\beta_2) = 1{,}000$ steps. A spike a hundred times the typical gradient keeps v above its usual level for more than two thousand steps, and throughout that time the inflated denominator $\sqrt{v}$ suppresses the updates of the coordinate. Clipping is therefore no substitute for lowering a learning rate that is too large for Adam, but it keeps one bad minibatch from silencing a parameter for thousands of steps. With the $\beta_2 = 0.95$ of Table 9.3, both effects shrink: the time constant falls to 20 steps, and the bound on the step falls to $0.1/\sqrt{0.05} \approx 0.45$ times η.

9.10.2.4 Loss Spikes and Recovery

Clipping does not prevent every failure. The PaLM run quoted at the start of this section had clipping enabled, and its loss still spiked about twenty times, at irregular intervals and sometimes late in training, while smaller models trained the same way did not spike. Other guards act on the loss rather than on the gradient. PaLM added a *z-loss*, the penalty $10^{-4}\log^2 Z$ on the softmax normalizer $Z = \sum_j \exp(o_j)$ of the output logits o_j. The cross-entropy does not change when the same constant is added to every logit, so nothing in it stops the logits from drifting as a whole toward large values, where they lose precision in bfloat16. The z-loss keeps $\log Z$ close to zero and with it the overall level of the logits. PaLM used it throughout training, and the spikes came anyway (Chowdhery et al. 2022).

When a spike does occur, the remedy is operational. The PaLM team restarted from a checkpoint about 100 steps before each spike and skipped the 200 to 500 minibatches seen around it, after which the loss did not spike again at that point. The skipped data was not bad in itself: the same minibatches, replayed from a different, earlier checkpoint, caused no spike. The spikes came from a particular combination of data and parameter state (Chowdhery et al. 2022). Recovery of this kind needs checkpoints frequent enough that a rewind costs hours rather than days, and each checkpoint must hold everything needed to resume: the model and optimizer state, the step count that drives the schedule, and the position in the data stream.

9.10.3 Weight Averaging

The last checkpoint of a run is one draw. At a constant learning rate, SGD does not settle at a point: its iterates keep fluctuating in a region around a minimum, the noise ball of Section 9.3, whose size grows with the learning rate and with the variance of the gradient noise. A schedule shrinks this region by decaying the learning rate (Section 9.6). Averaging the iterates reduces the noise in a different way, without touching the learning rate. When the fluctuations around the center of the region are only weakly correlated, they partly cancel in the mean.

9.10.3.1 The Exponential Moving Average

Stochastic weight averaging takes an equal-weight average of checkpoints from the tail of training (Izmailov et al. 2018). The exponential moving average (EMA) that Section 7.6 added to the standard training recipe weights recent iterates more heavily and needs no

choice of where the tail starts. It keeps a running average $\bar{\mathbf{x}}_t$ of the weights $\mathbf{x}_t$, written here with the decay α because β already denotes momentum:

$$\bar{\mathbf{x}}_t = \alpha\, \bar{\mathbf{x}}_{t-1} + (1-\alpha)\, \mathbf{x}_t. \tag{9.61}$$

This is the leaky average that momentum applies to gradients and Adam to squared gradients, now applied to the weights themselves. The weights of k steps ago enter with weight $(1-\alpha)\alpha^k$, so the average reaches back about $1/(1-\alpha)$ steps. Training never sees $\bar{\mathbf{x}}_t$; only evaluation uses it. Its cost is one more copy of the parameters in memory.

9.10.3.2 Averaging at a Constant Learning Rate

We test the average on the convolutional network `d2l.lenet_bn` of Section 9.6: a LeNet with ReLU activations, max-pooling, and batch normalization after every hidden layer, initialized with Xavier weights and zero biases, and trained on Fashion-MNIST in minibatches of 256 images. `d2l.evaluate_accuracy` of Section 9.5 measures its test accuracy. At the constant learning rate $\eta = 0.3$, the baseline of that section, its test accuracy gains little after about ten epochs and then moves up and down from epoch to epoch. Alongside the live weights we keep an EMA with $\alpha = 0.999$, whose window of about 1,000 steps spans roughly four epochs of 235 steps each.

```
fashion = d2l.FashionMNIST(batch_size=256)
device = d2l.try_gpu()
```

The averaged network needs batch-normalization statistics of its own for prediction, so we average the running means and variances along with the weights. This is an approximation: an average of the statistics of many networks is not the statistic of the averaged network. It is close when the weights change little over the window.

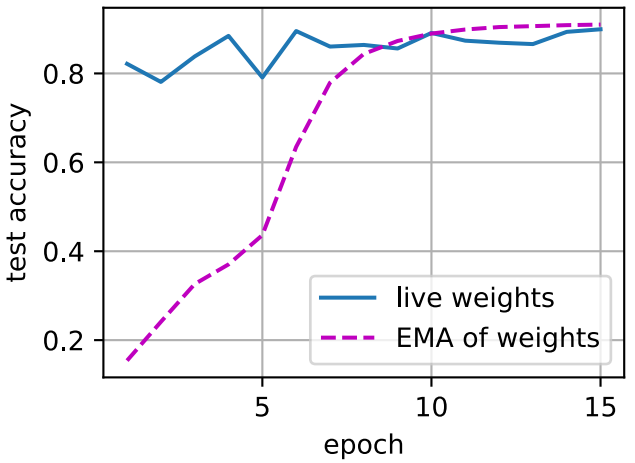
`AveragedModel` keeps a copy of the network. The first call of its method `update_parameters` copies the live weights into it, and with `get_ema_multi_avg_fn` every later call blends them in as in Equation 9.61. With `use_buffers=True` it averages the batch-normalization statistics too.

```
lr, num_epochs = 0.3, 15
net = d2l.lenet_bn().to(device)
ema = torch.optim.swa_utils.AveragedModel(
    net, multi_avg_fn=torch.optim.swa_utils.get_ema_multi_avg_fn(0.999),
    use_buffers=True)
ema.update_parameters(net)  # Start the average at the initial weights
optimizer = torch.optim.SGD(net.parameters(), lr=lr)
live_acc, avg_acc = [], []
for epoch in range(num_epochs):
    for X, y in fashion.train_dataloader():
        optimizer.zero_grad()
        nn.functional.cross_entropy(net(X.to(device)), y.to(device)).backward()
        optimizer.step()
        ema.update_parameters(net)
    live_acc.append(d2l.evaluate_accuracy(net, fashion.val_dataloader()))
    avg_acc.append(d2l.evaluate_accuracy(ema, fashion.val_dataloader()))
for name, acc in (('live', live_acc), ('EMA', avg_acc)):
```

```
    print(f'{name}: final test accuracy {acc[-1]:.3f}, last five epochs '
          f'{min(acc[-5:]):.3f} to {max(acc[-5:]):.3f}')
```

```
live: final test accuracy 0.899, last five epochs 0.866 to 0.899
EMA: final test accuracy 0.910, last five epochs 0.899 to 0.910
```

```
d2l.plot(list(range(1, num_epochs + 1)), [live_acc, avg_acc], 'epoch',
         'test accuracy', legend=['live weights', 'EMA of weights'])
```



Two features of the curves matter. First, the average needs its window to fill before it is useful. For the first few epochs it trails the live weights by a wide margin, because it still gives much weight to iterates near the initialization: after one epoch of 235 steps, the initial weights still carry the weight $0.999^{235} \approx 0.79$. This is lag, not the bias toward zero that Adam corrects (Section 9.5). The average starts at the initial weights, so its coefficients already sum to one, and there is nothing to correct. Practical EMAs therefore start averaging late in training or let α grow from a small value over the first steps.

Second, once the window has filled, the EMA curve is smooth and runs along the top of the live curve. Over the last five epochs the live accuracy jumps up and down, in some runs by several points, while the EMA's rises steadily, by about one to two points, and ends at its highest value. The live curve shows the noise ball directly: each epoch evaluates a different point in it, while the EMA evaluates close to its center.

9.10.3.3 Averaging and Decay

At a constant learning rate, the averaged weights reach about the final accuracy of the decayed schedules of Section 9.6, roughly 91%. Decay and averaging get there by different routes. Decay reduces the noise injected into future iterates and so changes the trajectory, while averaging leaves the trajectory alone and reduces the noise in the parameters we evaluate. The noise-ball argument also predicts a limit. Once a schedule has decayed the learning rate to nearly zero, the iterates barely move, and their average lies close to the last iterate. Averaging is therefore most useful when no decay is planned, or when a decayed endpoint is expensive to reach, as for the checkpoints evaluated during a long run whose final length is not yet fixed. Large runs average checkpoints even after decay, where the gain is smaller:

the released Llama 3 model is the average of checkpoints from the final phase of training, during which the learning rate was annealed to zero (Grattafiori et al. 2024). When you average a few saved checkpoints instead of keeping a running EMA, the batch-normalization statistics belong to none of the averaged weights. Recompute them with a pass over the training data before trusting the result.

9.10.4 Comparing Optimizers

Every comparison in this chapter rests on a tuning protocol, and the protocol is part of the result.

9.10.4.1 Scientific and Nuisance Hyperparameters

The Deep Learning Tuning Playbook (Godbole et al. 2023) sorts the hyperparameters of an experiment into three classes. *Scientific* hyperparameters are the ones the question is about. *Nuisance* hyperparameters must be re-optimized for every value of the scientific ones before the comparison means anything. *Fixed* hyperparameters are held constant, and each of them limits the claim. A comparison says something about its scientific hyperparameters only after the nuisance hyperparameters have been retuned in each arm. The most common violation is also the simplest. The learning rate is almost always a nuisance hyperparameter, and a single shared learning rate favors whichever method happens to prefer it.

The optimizer races of Section 9.5 and Section 9.7 followed this protocol. The optimizer was the scientific hyperparameter. The learning rate was the nuisance, retuned for each optimizer on a grid of four or five values spaced by factors of about three. The number of steps, the batch size, the model, and the absence of a schedule were fixed and stated, and every conclusion holds for that protocol only. At larger scale the same design has more nuisance dimensions (the peak learning rate, the decay horizon, the warmup, and the weight decay), replaces coarse grids by quasi-random search, and separates a broad exploration from a narrow final sweep.

9.10.4.2 Tuning Effort and the Verdict

The protocol can decide the verdict. Schmidt et al. (2021) benchmarked fifteen optimizers on a range of problems and found that trying several optimizers at their default settings works about as well as tuning a single one extensively. Defaults encode much accumulated tuning, and a comparison without per-arm tuning measures how much effort went into each arm rather than the algorithms. Wen et al. (2025) retuned ten optimizers for language-model pretraining, each with its own tuning budget, and found that published speedups of about $2\times$ over AdamW shrink to about $1.4\times$ at 100 million parameters and toward $1.1\times$ at 1.2 billion.

9.10.4.3 Experiment Records

A comparison is only as good as its record. For every run, log the full configuration, including the hyperparameters you consider fixed, the code version, the random seed, and the one thing you changed. Change one factor at a time to estimate its effect, and use a factorial design when the interaction between factors is the question. Repeat the decisive runs with new random seeds: a difference smaller than the spread between seeds is not a result. Keep the runs that diverged, since the edge of a sweep where runs fail marks the

stability boundary, as the divergent grid point of Section 9.5 did for SGD. And write the conclusion next to the curves while the experiment is fresh. A directory of loss curves without sentences becomes unreadable within weeks.

9.10.4.4 The Chapter's Three Decisions

The methods of this chapter settle three decisions: the direction of an update (the gradient, its sign, or an orthogonalized version of it), the size of the step over time (warmup, decay, and the shape of the schedule), and the handling of gradient noise (the batch size, momentum, and averaging). Every row of Table 9.3 is one coordinated choice of all three. When a run misbehaves, the first question is which of the three decisions failed. This chapter has left open where the updates run once a model no longer fits on one device, and how the optimizer state is divided among many devices. That is the subject of Chapter 13.

9.10.5 Summary

A large training run is configured once, guarded against rare failures, and judged by its final weights. Recent pretraining reports disclose recurring settings (AdamW with $(\beta_1, \beta_2) = (0.9, 0.95)$, weight decay 0.1, clipping at 1, warmup and a later decay) that make a defensible start, and with a small budget the peak learning rate is the setting to tune. Global-norm clipping with a threshold above the typical gradient norm fires only on the first few steps, where the gradients of the fresh initialization are largest. Yet it turns a diverging `TinyLM` run at $\eta = 1$ into a stable one, acting as an automatic warmup, and in long runs it also catches the rare oversized gradients of heavy-tailed noise. Spikes that get through are handled by rewinding to a checkpoint and skipping the data seen around them. An exponential moving average of the weights reduces the noise in the evaluated parameters without changing the learning rate, and at a constant rate it reaches the accuracy of a decayed schedule. A comparison of optimizers is evidence only when each arm's nuisance hyperparameters were retuned and the protocol is stated.

9.10.6 Exercises

1. **The decay timescale.** The horizon of Section 9.5.4.4 is $\tau = B/(\eta\lambda D)$ epochs for batch size B and dataset size D, both in tokens. The DeepSeek-V3 and OLMo 2 reports in Table 9.3 use sequences of 4,096 tokens and training sets of about 14.8 trillion and 4 trillion tokens.
    1. Compute τ for both rows at their peak learning rates, taking for DeepSeek-V3 its final batch size of 15,360 sequences. What fraction of each dataset does the averaging horizon span?
    2. Following Bergsma et al. (2025a), what should happen to λ if the batch size were doubled and τ is to be preserved, and what if instead the dataset were doubled?
2. code **Clipping regimes.** The demonstration in Section 9.10.2.1 clips at $\theta = 1$.
    1. Implement global-norm clipping Equation 9.60 yourself as a function that rescales the gradients by $\min(1, \theta/(\|\mathbf{g}\|_2 + 10^{-6}))$ and returns the norm before clipping. Check on one minibatch that it returns the same norm as the library function and clipped gradients that agree with the library's to about 10^{-6}.
    2. Sweep $\theta \in \{0.01, 0.1, 0.5, 1, 4, \infty\}$ on the divergent run, recording the final loss and the fraction of steps on which clipping changed the update. Classify each

threshold as a guard (fires rarely), a brake (fires on most steps), or absent, and relate each class to whether the run diverges and to its final loss.

3. Show that for SGD without momentum, clipping that fires on every step turns the update into normalized gradient descent with step size $\eta\theta$. The divergent run uses momentum, with velocity $\mathbf{v}_t = \beta\mathbf{v}_{t-1} + \mathbf{g}_t$, update $\mathbf{x}_t = \mathbf{x}_{t-1} - \eta\mathbf{v}_t$, and $\beta = 0.9$. Bound the length of one of its updates when clipping fires on every step. Explain why neither case is the same as training at a lower learning rate.
4. In the demonstration, clipping fired only on the first steps. Remove it and instead warm the learning rate up linearly from 0 to $\eta = 1$, once over the first 20 steps and once over the first 100 steps. Does each run survive? Compare the warmup lengths with the last step on which clipping fired, and explain what the two remedies have in common and how they differ.

Adapted from Stanford CS336, Assignment 1, Problem gradient_clipping.

3. **Logit drift.** The z-loss of PaLM (Chowdhery et al. 2022) adds $10^{-4}\log^2 Z$ to the cross-entropy, where $Z = \sum_j \exp(o_j)$ is the softmax normalizer of the output logits $\mathbf{o}$.

   1. Show that adding the same constant c to every logit leaves the cross-entropy unchanged, and compute how the shift changes $\log Z$ and the z-loss.
   2. Compute the gradient of $10^{-4}\log^2 Z$ with respect to $\mathbf{o}$. In which direction does a gradient step move the logits when $\log Z$ is large and positive? Show that the gradient of the cross-entropy alone has no component along the all-ones vector, and explain what this implies for the overall level of the logits during training.

4. code **The averaging window.** Sweep the EMA decay $\alpha \in \{0.9, 0.99, 0.999, 0.9999\}$ in the demonstration of Section 9.10.3 and plot the EMA's test accuracy per epoch for each value.

   1. Express each averaging window $1/(1-\alpha)$ in epochs. Derive the total weight that the EMA still places on the initial weights after the $T \approx 3{,}500$ steps of the 15-epoch run, and evaluate it for each α.
   2. Use these numbers to explain the curves at the two ends of the sweep.
   3. Add the EMA with $\alpha = 0.999$ to a run whose learning rate decays to zero along the cosine schedule of Section 9.6. Does the average still improve on the final live weights? Explain the answer with the noise ball.

5. code **A ten-run budget.** Using AdamW on `TinyLM`, reach a training loss of 1.1 in as few steps as you can, tuning only the learning rate, with a budget of ten runs. A run reaches the target at the first step at which the mean of its last 100 training losses falls below 1.1. Cap each run at 2,000 steps. Keep the log that Section 9.10.4 asks for: for every run, the learning rate, the steps taken, the final loss, and the one-sentence conclusion you drew before launching the next run. Report the log, not just the winning configuration.

6. **Defaults against a small budget.** Schmidt et al. (2021) found that trying several optimizers at default settings works about as well as tuning a single one. Test this on `TinyLM` with runs that Section 9.5 and Section 9.7 already made under the same protocol. As the default runs, take SGD with momentum 0.9 at $\eta = 0.1$, Adam at $\eta = 10^{-3}$, and the Muon/AdamW hybrid at $\eta = 10^{-3}$, the rate that RMS matching makes comparable to Adam's default.

   1. Compare the best of the three default runs with the best point of each of the four grids: SGD and Adam in Section 9.5, AdamW and the Muon/AdamW hybrid in

Section 9.7. Which optimizers, tuned alone, beat the best default run, and by how much in final loss?
2. Given a fixed budget of four runs on an unfamiliar problem, which strategy would you choose, and on what does the answer depend?

7. code **Batch size and the tuned rate.** ▯ Rebuild `data` with a batch size of 256 instead of 64. At both batch sizes, train AdamW without weight decay on `TinyLM` from the same initialization at a constant learning rate, sweeping $\eta \in \{0.5, 1, 2, 4, 8, 16, 32\} \times 10^{-3}$. Give both batch sizes the same number of training sequences: 2,000 steps at batch size 64 and 500 steps at batch size 256.
   1. Where does the optimum move? Compare the shift with the two rules of Section 9.8, the square-root rule $\eta \propto \sqrt{b}$ and the linear rule $\eta \propto b$.
   2. What would you have concluded about AdamW at batch size 256 had you reused the batch-64 learning rate without retuning?

9.11 Further Reading

The sections of this chapter teach one representative of each idea: one valley, one stochastic method, one way to average over time, one adaptive method, one matrix optimizer, and one set of rules for batch size and width. This section points to topics they had no room for: other views of the loss surface, the classical methods that deep learning rarely uses, further preconditioners and faster ways to orthogonalize, other schedules and averages, and the evidence from large training runs and benchmarks. Each topic states the problem it addresses and what it changes relative to the chapter. Implementations, tutorials, and courses for the material that the chapter does teach are listed in its introduction, Chapter 9. Nothing else in the book depends on this section.

9.11.1 The Loss Surface

9.11.1.1 Pictures of a High-Dimensional Loss

Section 9.1 drew its landscapes in one and two dimensions, where every direction can be shown. A network has millions, and the common remedy is to plot the loss along one or two random directions through the trained weights. Such plots mislead unless the directions are rescaled to the size of the weights they perturb: a ReLU network computes the same function after one layer is multiplied by c and the next by $1/c$, and an unnormalized direction then looks sharp in one network and flat in its rescaled twin. H. Li et al. (2018) normalize each random direction filter by filter to the norm of the corresponding filter. With this normalization, deep networks without skip connections show chaotic, nonconvex surfaces, and the same networks with the residual connections of Section 7.4 show smooth, nearly convex basins.

9.11.1.2 Connected Minima

The permutation argument of Section 9.1 showed that a network has many minima that compute the same function, and the average of two of them is usually worse than either. Minima reached from different random starts, not related by a permutation, also look isolated along the straight line between them, where the loss rises in between. Garipov et al. (2018) and Draxler et al. (2018) found that such minima are nonetheless joined by simple

curves, a polygonal chain with one bend or a quadratic Bézier curve, along which the training and test loss stay low. The barrier on the straight line is then largely the permutation symmetry itself: Entezari et al. (2022) conjectured that once the hidden units of one network are permuted to match the other, the straight line has almost no barrier, and Ainsworth et al. (2023) found such permutations by matching weights or activations and confirmed the conjecture for wide networks. This alignment is the basis of methods that merge separately trained models by averaging their weights.

9.11.1.3 Flat Minima and Sharpness-Aware Minimization

A long-standing hypothesis holds that minima in flat regions generalize better than sharp ones, because a small perturbation of the weights barely changes the loss there. Sharpness-aware minimization (Foret et al. 2021) turns the hypothesis into an optimizer. Before each descent step it takes a step of fixed length up the gradient, and it applies the gradient computed at that perturbed point to the original weights. It thereby minimizes the largest loss in a small ball around the weights rather than the loss at one point, at twice the gradient cost per step. The reported gains are largest in image classification and fine-tuning. The premise itself is debated: Dinh, Pascanu, et al. (2017) showed with the rescaling of the previous paragraphs that a minimum of a ReLU network can be made arbitrarily sharp without changing the function it computes, so that sharpness, unlike generalization, depends on how the function is parametrized.

9.11.2 Classical Methods

9.11.2.1 Quasi-Newton Methods

Newton's method of Section 9.2 needs the Hessian and a linear solve with it at every step. Quasi-Newton methods build an approximation of the inverse Hessian from the differences between successive gradients and iterates, so they need only gradients. BFGS updates a dense $d \times d$ matrix, and limited-memory BFGS (Liu and Nocedal 1989) keeps only the last m pairs of differences and applies the approximation implicitly, at $O(md)$ memory instead of $O(d^2)$. L-BFGS is the default method for smooth, deterministic problems of moderate size, such as logistic regression on a dataset that fits in memory. It is rare in deep learning because minibatch noise corrupts the gradient differences from which it estimates curvature. Section 35.2.2.2 derives the BFGS update, and Nocedal and Wright (2006) treat the whole family.

9.11.2.2 Line Search and Trust Regions

The chapter fixes the learning rate or schedules it in advance. A *line search* chooses it anew at every step: it tries a step along the descent direction, accepts it if the objective has fallen by at least a fixed fraction of what the gradient predicts (the Armijo condition), and shrinks it otherwise. The step then adapts to the local curvature without knowing it, but each trial costs an evaluation of the objective, which for a training loss is a pass over the data, and a minibatch estimate makes the test itself noisy. A *trust-region method* bounds the length of the step instead of scaling the gradient: it minimizes the quadratic model of the loss inside a ball, and grows the ball when the model predicted the decrease well and shrinks it when it did not. Inside a ball, negative curvature is harmless. This repairs the failure of the raw Newton step of Section 9.2, which climbed to a local maximum. The appendix treats both (Section 35.1.1 and Section 35.2.3), as do Nocedal and Wright (2006) and Boyd and Vandenberghe (2004) .

9.11.2.3 Newton Steps Without the Hessian

A network's Hessian cannot be stored, but its product with a vector costs about as much as two gradients. Hessian-free optimization (Martens 2010) uses this to take approximate Newton steps: it solves $\mathbf{H}\epsilon = -\nabla f$ by a few iterations of conjugate gradients, which need only such products, and damps the system to keep the step safe. It trained deep autoencoders without the layer-by-layer pretraining that was common at the time. Saddle-free Newton (Dauphin et al. 2014) addresses the other weakness of Newton's method, its attraction to saddle points and maxima. It applies the remedy that Section 9.2 mentions, the absolute value of the Hessian, the matrix with the same eigenvectors and the magnitudes of its eigenvalues, so that a direction of negative curvature repels the iterate instead of attracting it. For a network it computes this matrix on a small subspace spanned by Hessian–vector products. The exercise of Section 9.2 on absolute-curvature preconditioning is the diagonal version of the idea.

9.11.2.4 Rates for Stochastic Gradient Descent

Section 9.3 stated the conditions of Robbins and Monro (Robbins and Monro 1951) under which SGD converges and measured its noise ball, but quoted no rates. For convex objectives with bounded gradients, SGD with learning rates proportional to $1/\sqrt{t}$ reaches an error of order $1/\sqrt{T}$ after T steps if its iterates are averaged (Polyak and Juditsky 1992), and no method that sees only stochastic gradients can do better in the worst case (Nemirovski and Yudin 1983). For smooth nonconvex objectives, where a minimum cannot be guaranteed, SGD drives the expected squared gradient norm to zero at the same order (Ghadimi and Lan 2013). The appendix proves both results (Section 35.4.3 and Section 35.4.3.2). Bottou et al. (2018) survey the theory and practice of stochastic methods for large-scale learning, from noise and averaging to second-order methods.

9.11.2.5 Variance Reduction

On a finite training set, the noise of SGD can be removed at the minimum itself. Methods of the SVRG family store a full gradient at a reference point and correct each stochastic gradient by the difference between the gradients of its examples at the current and the reference point. The corrected estimate is still unbiased, its variance shrinks as both points approach the minimum, and on a strongly convex objective a constant learning rate then converges at the linear rate of gradient descent. The theory is clean, but on deep networks these methods have not given consistent improvements over SGD with momentum. Section 35.4.4 derives them and discusses why.

9.11.3 Adaptive and Matrix Preconditioners

9.11.3.1 A Method Without a Learning Rate

Adadelta (Zeiler 2012) adds a second exponential average to RMSProp, of the squared *updates*, and replaces the learning rate by the square root of that average, so that the numerator and the denominator of the update carry the same units. The method has no learning rate in name. In practice its scale on the first steps comes from ϵ, which then acts as one.

9.11.3.2 Kronecker-Factored Preconditioners

Adam's preconditioner of Section 9.5 is diagonal, and Shampoo's in Section 9.7 keeps one small matrix per side of each weight matrix. K-FAC (Martens and Grosse 2015) arrived at a preconditioner of the same shape from a different question. The *natural gradient* (Amari 1998) preconditions with the inverse of the Fisher information matrix, the covariance of the gradients of the log-likelihood, which measures the size of a step by how much it changes the model's predictions rather than its weights. K-FAC approximates the Fisher block of a layer by the Kronecker product of two small matrices, the second moments of the layer's inputs and of the gradients with respect to its outputs, and so needs the inverses of two small matrices instead of one huge one. SOAP (Vyas et al. 2024) runs Adam in the eigenbasis of Shampoo's two matrices. Because Adam's statistics adapt within that basis, the expensive eigendecompositions need to be recomputed only occasionally. SOAP is among the fastest methods in the matched-tuning studies cited in Section 9.7.

9.11.3.3 Steepest Descent Under a Chosen Norm

Section 9.7 derived Muon as steepest descent under the spectral norm and gave the remaining parameters to AdamW. Scion (Pethick et al. 2025) assigns a norm to every layer instead: the spectral norm for hidden matrices, a norm that normalizes each column for embeddings with one-hot inputs, and a sign or row-normalized step for the output layer. Each step is then the minimizer of a linear function over a ball of that norm, and the resulting optimizer needs no Adam at all. In the same view, decoupled weight decay keeps the weights inside a ball of the norm, and the learning rates transfer across widths. Kovalev (2025) reads Muon as a trust-region method in a non-Euclidean norm, which explains why it needs weight decay. On where Muon's gain over Adam comes from, S. Wang et al. (2025) find that it concentrates in the value and output projections of attention and in the MLP weights, and, in an associative-memory model, on rare classes, the heavy-tailed regime in which Section 9.5 located Adam's own advantage over SGD.

9.11.3.4 Faster Orthogonalization

Newton–Schulz in Section 9.7 adds about an eighth to the arithmetic of a training step on `TinyLM`, and several lines of work reduce this cost. Before Polar Express, six quintics found by a heuristic search, one per step (Cesista et al. 2025), improved the speedrun times of the next paragraph by one to two percent. CANS (Grishina et al. 2025), concurrent with Polar Express, uses Chebyshev alternation and the Remez algorithm to find polynomials that map the spectrum into an interval $[1-\delta, 1+\delta]$ with the largest possible slope at zero. Gram Newton–Schulz (Zhang et al. 2026) uses that every odd polynomial step has the form $h(\mathbf{X}\mathbf{X}^\top)\mathbf{X}$, so the whole composition can run on the small Gram matrix with symmetric products and only two multiplications by the rectangular matrix. For rectangular weights this is up to about twice as fast. Turbo-Muon (Boissin et al. 2025) rescales the rows of the momentum by a diagonal matrix that makes it nearly orthogonal before the iteration starts, which saves one of the five steps. On the theory side, Kim and Oh (2026) show that Muon with q Newton–Schulz steps converges at the rate of exact orthogonalization up to a constant factor that approaches one doubly exponentially fast in q, so that a few steps lose almost nothing against exact orthogonalization.

9.11.3.5 Distributed Orthogonalization

Dion, in Section 9.7, avoids gathering a sharded matrix by orthogonalizing a low-rank approximation. Its successor Dion3 (Amsel, Zhang, et al. 2026) orthogonalizes at each step only a fraction, a quarter or an eighth, of the rows of the momentum, those with the largest ℓ_1 norms, keeps the rest in the buffer as Dion does, and can add NorMuon's per-neuron scaling. On models with a billion parameters under fully sharded data parallelism, the authors report Muon's loss with up to six times less time spent in the optimizer. MuonBP (Khaled et al. 2025) orthogonalizes each device's shard on its own and the full matrix only every few steps, with a separate learning rate for each kind of step.

9.11.3.6 Smaller and Sharded Optimizer State

Adam keeps two numbers per parameter, which for a large model costs more memory than the weights. Adafactor (Shazeer and Stern 2018) stores for each matrix only the row and column sums of $\mathbf{v}$, $m + n$ numbers instead of mn, rebuilds an estimate of $\mathbf{v}$ from their outer product, and can drop the first moment. 8-bit optimizers (Dettmers et al. 2022) keep both moments but quantize them block by block to one byte each, with a scale per block, which preserves the accuracy of 32-bit Adam on the workloads tested. ZeRO (Rajbhandari et al. 2020) reduces the state per device instead of the state in total: it partitions the optimizer state, then the gradients, then the weights across the devices of a data-parallel group, rather than replicating them on each. The updates are those of this chapter. Where their bytes live and when they move is the subject of Chapter 13, in particular Section 13.6.3.

9.11.4 Schedules and Averages

9.11.4.1 Warm Restarts

Cosine decay, as used in Section 9.6, was proposed together with periodic restarts (Loshchilov and Hutter 2016): at the end of each cosine cycle, the learning rate jumps back to its peak, and the cycles may grow in length. Each cycle ends with a decayed model, so one run yields several finished models, which can be ensembled. Warmup–stable–decay obtains models at several budgets from one run differently, by branching decays off a single plateau without ever returning to a high rate.

9.11.4.2 Warmup and Representations

Section 9.6 explained warmup through the sharpness of the loss at initialization. Gotmare et al. (2018) asked instead what warmup changes inside the network. They compared the representations of each layer during training with and without warmup and found that its main measurable effect is to limit how far the weights of the deepest layers move while the parameters and the optimizer's statistics are still near their random initial values.

9.11.4.3 Schedule-Free Optimization

Decay removes the noise ball at the end of training by shrinking the late steps. Averaging the iterates, as in Section 9.10, removes it instead from the parameters that are evaluated. Schedule-free SGD and AdamW (Defazio et al. 2024) build training around the average: they keep a constant learning rate, evaluate each gradient at an interpolation between the current iterate and the running average of all iterates, and return the average. The average

needs no training horizon, which gives it the operational advantage of warmup–stable–decay: training can stop, or continue, at any step. The method won the self-tuning track of the AlgoPerf benchmark described at the end of this section. An exercise of Section 9.6 implements its recurrence on the noisy valley.

9.11.4.4 Averaging Checkpoints and Models

Section 9.10 averaged the iterates of one run with an exponential moving average. Three variants change what is averaged. LAWA (Kaddour 2022) averages the latest few saved checkpoints with equal weights, and on ImageNet and BERT training the average reached the accuracy or loss of a single checkpoint taken considerably later. Model soups (Wortsman et al. 2022) average the weights of several models fine-tuned from the same pretrained model with different hyperparameters. The soup often beats the best single model and adds no cost at inference. It works because fine-tuned models from a shared start lie in one basin. Diffusion models (Section 17.3) evaluate an average of their weights as a matter of course, and their sample quality depends strongly on the averaging window. Karras, Aittala, Lehtinen, et al. (2024) store a few snapshots during training and reconstruct from them, after training, an exponential average of any length, so that the window can be tuned without training again.

9.11.5 Large Runs and Benchmarks

9.11.5.1 Muon from the Speedrun to Production

Muon was first demonstrated on the NanoGPT speedrun (Jordan and contributors 2024; Jordan et al. 2024), a public competition to train a fixed GPT-2-sized model to a fixed validation loss in the least wall-clock time, in which each record changes one thing and publishes the change. Muon's first record cut the time by about a fifth. The headline time has since fallen far further, from changes to the optimizer, the architecture, the systems, and the data together, so it is a record of engineering rather than a benchmark of optimizers. Moonlight (Jingyuan Liu et al. 2025) was the first Muon run with billions of parameters. It introduced the RMS-matching factor of Section 9.7 and decoupled weight decay for Muon, and found that without decay, Muon falls behind AdamW late in long runs as the weight norms grow. Kimi K2 (Kimi Team 2025a) and GLM-4.5 (Zeng et al. 2025) were pretrained with Muon.

9.11.5.2 Attention-Logit Stability

The logits of attention are dot products of queries and keys, so they grow with the norms of the vectors that the projections produce, and large logits saturate the softmax, a common source of loss spikes in transformers. QK-norm (Henry et al. 2020) normalizes the queries and keys just before their dot product, which bounds the logits whatever the norms upstream. OLMo 2 (Team OLMo et al. 2025) adopted it among the stability changes its report documents. The QK-clip of Section 9.7 reaches the same bound differently: it leaves the forward pass alone and rescales the projection weights after an optimizer step.

9.11.5.3 Layerwise Adaptive Learning Rates

LARS and LAMB rescale the update of each layer by the ratio of the norm of its weights to the norm of its update, a rule proposed to make very large batches trainable. When Nado et al. (2021) retuned the learning rate, warmup, and other nuisance hyperparameters of the

baselines as carefully, plain momentum and Adam matched them at the batch sizes studied. The comparison illustrates the point of Section 9.10: a new method must be compared against an equally well-tuned baseline.

9.11.5.4 Hyperparameter Transfer in Production

Section 9.9 transferred a learning rate across widths of one MLP. The Cerebras-GPT family (Dey et al. 2023) applied muP, including its transformer-specific rules, to a whole model family, with hyperparameters transferred from a proxy of about 40 million parameters, and published its coordinate checks with the models. DeepSeek (DeepSeek-AI, Bi, et al. 2024) kept the standard parametrization and predicted the drift instead: it fitted power laws for the best learning rate and batch size as functions of the compute budget across many small runs and extrapolated them, the approach of an exercise in Section 9.9. Meta reports that Llama 4 (Meta AI 2025) was trained with per-layer learning rates and initialization scales that transfer across width, depth, batch size, and training length, without publishing the method. Transfer in production now extends beyond width, the one direction the chapter covers.

9.11.5.5 Benchmarks and Their Protocols

How optimizers are compared decides which one wins. The AlgoPerf benchmark (Dahl et al. 2023) measures the time to reach a target on a fixed set of workloads, with hyperparameters either tuned within a fixed budget of trials or set by the method itself. In its first competition (Kasimbeg et al. 2025), distributed Shampoo won the externally tuned track and schedule-free AdamW the self-tuning track. The open-ended tinkering of the speedrun favored Muon instead. Semenov et al. (2025) ran a second matched-tuning benchmark on dense and mixture-of-experts language models: AdEMAMix, which adds a slow second momentum average to AdamW, was consistently strong, Muon with weight decay was strong especially in short runs, and Muon without decay on its matrices fell behind. The study of Wen et al. (2025) cited in Section 9.10 also found that rankings can flip late in a run as the learning rate decays, so optimizers should be compared at the end of training rather than along the way. The modded-NanoGPT optimization track (Jordan 2026) fixes the model, the data, and the batch, counts steps to a target loss over many seeds, and accepts a record only if the difference is significant. On it, a tuned Muon ties NorMuon.

9.11.5.6 A Training Logbook

Reports of large runs give the final recipe; the work of keeping a run alive is rarely published. The OPT team (Zhang et al. 2022) released its logbook with its models: two months of restarts after loss spikes and hardware failures, and of learning rates lowered in mid-run. It complements the rewind-and-skip remedy of Section 9.10 with a day-by-day record of what such remedies cost.

Chapter 10

Attention

The sequence models of Chapter 8 read a text one token at a time and carry what they have read in a state of fixed size. Think of a reader who reaches the last chapter of a mystery novel and needs the name of a minor character from the first page. A person turns back and looks. A recurrent network must have kept the name in its state all along, written when nothing marked it as important. Can a network look back instead, and read only what the current prediction needs?

Attention does this. It keeps a representation of every position, scores the current query against a key at every position, turns the scores into weights, and returns the weighted average of the values stored there. The terms *query*, *key*, and *value* come from database lookup, and the weighted average makes the lookup differentiable. The idea came from machine translation, where a decoder had to work from a single vector summarizing the source sentence. Attention let it consult every source word instead (Bahdanau et al. 2015). The Transformer then made attention the only operation that mixes positions (Vaswani et al. 2017), and it is now the core of large language models.

Reading every position raises problems that recurrent networks never had. A weighted average ignores the order of its inputs. Scoring all pairs of positions costs time and memory that grow with the square of the length. And a map of attention weights shows where a head looks, not what it computes.

Section 10.1 asks how a lookup can be made differentiable, starting from classical kernel regression. Section 10.2 asks which score a network should learn, and how masks keep a query away from padding and from the future. Section 10.3 asks what one weighting per query cannot express, and what several heads, and heads that share keys and values, provide. Section 10.4 asks how an operation that is blind to order learns where each token stands, and what happens beyond the length it was trained on. Section 10.5 asks how to cut the cost of scoring all pairs of positions: by exact attention in less memory, by a window of recent keys, or by a recurrence of linear cost. Section 10.6 asks what trained heads compute, and which experiments can tell.

The chapter studies the attention layer by itself. Chapter 11 adds the feed-forward layers, normalization, and training recipe that make it a transformer, and Chapter 12 develops the recurrent side of linear attention.

Resources and Further Reading

The resources below accompany the sections of this chapter; each names the section it goes with. All are freely accessible online. Related topics that the chapter does not teach are collected in Section 10.7.

Visual introductions

- Attention in Transformers, Step-by-Step — 3Blue1Brown (2024) — a geometric introduction to queries, keys, values, and attention weights, the lookup of Section 10.1 drawn as vectors in an embedding space.
- The Illustrated Transformer — Jay Alammar (2018) — pictures of scaled dot-product and multi-head attention with the shapes of every matrix, the computation of Section 10.2 and Section 10.3.
- Transformers from Scratch — Peter Bloem (2019) — derives self-attention as an operation on sets, with the permutation equivariance that Section 10.4 proves and then breaks.

Foundational papers

- Neural Machine Translation by Jointly Learning to Align and Translate — Bahdanau et al. (2014) — the first learned alignment, whose additive score Section 10.2 compares with the dot product.
- Attention Is All You Need — Vaswani et al. (2017) — scaled dot-product attention, multi-head attention, and sinusoidal positions, the starting point of Section 10.2 through Section 10.4.
- Fast Transformer Decoding: One Write-Head Is All You Need — Shazeer (2019) — multi-query attention, the extreme case of the shared keys and values of Section 10.3, motivated by the memory traffic of decoding.
- RoFormer: Enhanced Transformer with Rotary Position Embedding — Su et al. (2021) — rotary embeddings, derived as in Section 10.4 from the requirement that scores depend only on offsets.
- Train Short, Test Long — Press et al. (2022) — ALiBi and the length-extrapolation experiment that Section 10.4 repeats on a small model.
- YaRN: Efficient Context Window Extension of Large Language Models — Peng et al. (2023) — position interpolation, the NTK-aware base change, and YaRN's treatment of fast and slow pairs, the context extension of Section 10.4.

The cost of attention

- FlashAttention: Fast and Memory-Efficient Exact Attention with IO-Awareness — Dao et al. (2022) — the tiled online softmax of Section 10.5, with the memory-traffic analysis that explains its speed.
- Transformers Are RNNs — Katharopoulos et al. (2020) — linear attention and its recurrent form, the third remedy of Section 10.5.
- The Transformer Family Version 2.0 — Lilian Weng (2023) — a survey of attention variants, sparse patterns, and efficient approximations around Section 10.5.

What attention computes

- A Mathematical Framework for Transformer Circuits — Elhage et al. (2021) — the residual stream and the QK and OV circuits that Section 10.6 reads from trained weights.
- In-Context Learning and Induction Heads — Olsson et al. (2022) — induction heads in small and large models, the circuit that Section 10.6 trains and dissects.

- Thinking Like Transformers (Transformer Puzzles) — Sasha Rush — puzzles in RASP, a language for what fixed-depth attention can compute, a companion to the expressivity arguments of Section 10.6.

Courses

- Stanford CS224n: Natural Language Processing with Deep Learning — lectures and an assignment on self-attention, positions, and transformer components, the material of Section 10.2 through Section 10.4.

10.1 Queries, Keys, and Values

The recurrent networks of Section 8.4 keep everything they know about the past in a state of fixed size, and every prediction is made from that state. The network has to decide what to write there before it knows what a later prediction will need. Consider a recurrent language model as wide as GPT-2 small that has read 1,024 tokens. Its state holds 768 numbers. If the next prediction depends on the 17th token, it must be made from those 768 numbers, all written before the network could know that the 17th token would matter. Convolutions over sequences (Kalchbrenner et al. 2014) face the same limit, and on long inputs the fixed-size summary loses information that a later prediction needs. Can a model keep its whole input and read from it only the part that each prediction uses?

A database does this. It stores its records as they are and answers each query by comparing it with the stored keys. An exact lookup, however, either finds a matching key or returns nothing, and a network trained by gradient descent needs an answer that changes smoothly with the query. Consider the scalar database $\mathcal{D} = \{(2, 10), (5, 20)\}$ of (key, value) pairs. The exact query $q = 5$ returns the value 20, and the query $q = 4$ returns nothing at all, because no key matches. A *soft* query instead scores every key, for example by $a(q, k) = -(q - k)^2$, and turns the scores into weights that sum to one. For $q = 4$ the scores are -4 for key 2 and -1 for key 5. The softmax turns them into the weights $e^{-4}/(e^{-4}+e^{-1}) \approx 0.05$ and 0.95, so the answer is $0.05 \cdot 10 + 0.95 \cdot 20 \approx 19.5$. The answer moves smoothly with the query, the same rule works for a database of any size, and nothing has to be compressed before the query arrives.

The *attention mechanism* (Bahdanau et al. 2015) is this soft lookup, used as a layer of a neural network. This section defines it and studies it in its oldest form, kernel regression, where the weights come from a fixed similarity function rather than from learning. The next section makes the scores learnable.

10.1.1 Attention as Differentiable Lookup

The soft query of the example used scalars and two stored pairs. In a network the keys and values are vectors, and there is a pair for every position of the input, so we now write the same rule in that generality.

Let the query and the keys be vectors $\mathbf{q}, \mathbf{k}_i \in \mathbb{R}^d$ and the values $\mathbf{v}_i \in \mathbb{R}^{d_v}$, and let $\mathcal{D} \stackrel{\text{def}}{=} \{(\mathbf{k}_1, \mathbf{v}_1), \dots, (\mathbf{k}_m, \mathbf{v}_m)\}$ be a database of m key–value pairs. The attention output is the weighted sum of the values

$$\text{Attention}(\mathbf{q}, \mathcal{D}) \stackrel{\text{def}}{=} \sum_{i=1}^{m} \alpha(\mathbf{q}, \mathbf{k}_i)\mathbf{v}_i, \tag{10.1}$$

where the scalar weight $\alpha(\mathbf{q}, \mathbf{k}_i)$ says how much value i contributes. In every softmax attention layer of this book the weights are nonnegative and sum to one, so the output is a convex combination of the values: it lies inside their convex hull. Exact lookup is the special case in which one weight is 1 and the others are 0. Uniform weights $1/m$ give the plain average of the values, *average pooling*.

Weights with these two properties come from any scoring function $a(\mathbf{q}, \mathbf{k})$ by exponentiating the scores, which makes them positive, and dividing by their sum. This is the softmax operation of multinomial logistic regression (Section 3.1):

$$\alpha(\mathbf{q}, \mathbf{k}_i) = \frac{\exp(a(\mathbf{q}, \mathbf{k}_i))}{\sum_j \exp(a(\mathbf{q}, \mathbf{k}_j))}. \tag{10.2}$$

The softmax is differentiable, so gradients reach the query, the keys, and the values. For finite scores and at least two keys its Jacobian is never the zero matrix, although its entries become very small when one score dominates the others. Attention can also make hard, nondifferentiable choices. It must then be trained with reinforcement learning (Mnih et al. 2014) and is harder to optimize, so this book uses the differentiable form throughout. Figure 10.1 works one lookup through with three keys.

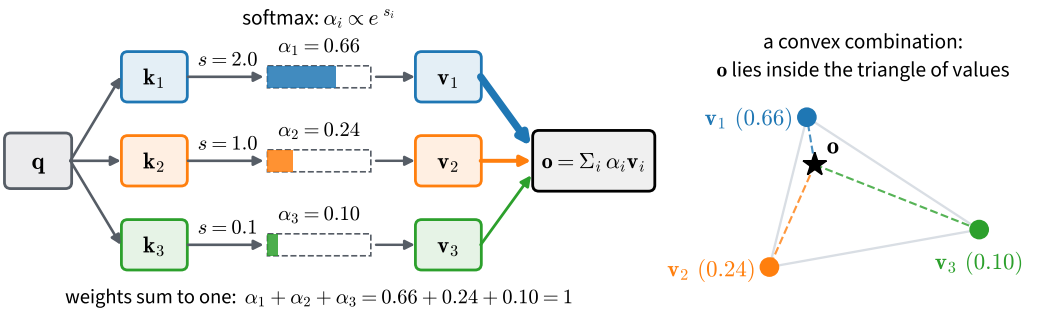


Figure 10.1: Soft lookup with three key–value pairs. The query's scores s_i against the keys become softmax weights α_i that sum to one (left), and the output $\mathbf{o}$ is the weighted sum of the values, a point inside the triangle whose corners are the values (right).

The scoring function has a fixed number of parameters, however many key–value pairs the database holds. One attention layer therefore serves inputs of every length, and it reads all of them directly instead of a compressed summary. What grows instead is memory. A recurrent network keeps one state, while attention keeps a key and a value for every position it may need to read.

10.1.2 Attention Pooling with Fixed Kernels

The weights in Equation 10.1 need not be learned. Long before deep learning, statisticians used the same weighted sum with a weight function chosen by hand, in Nadaraya–Watson kernel regression (Nadaraya 1964; Watson 1964). On a problem small enough to plot, this classical setting shows what the weights do. Its limits show why the rest of the chapter learns them.

10.1.2.1 Similarity Kernels

A *kernel* $\alpha(\mathbf{q}, \mathbf{k})$ measures the similarity of a query and a key, typically as a decreasing function of their distance. Common choices are

$$\begin{aligned} \alpha(\mathbf{q}, \mathbf{k}) &= \exp\left(-\tfrac{1}{2}\|\mathbf{q} - \mathbf{k}\|^2\right) && \text{Gaussian;} \\ \alpha(\mathbf{q}, \mathbf{k}) &= 1 \text{ if } \|\mathbf{q} - \mathbf{k}\| \le 1 && \text{boxcar;} \\ \alpha(\mathbf{q}, \mathbf{k}) &= \max\left(0, 1 - \|\mathbf{q} - \mathbf{k}\|\right) && \text{triangular,} \end{aligned} \tag{10.3}$$

and the degenerate constant kernel $\alpha(\mathbf{q}, \mathbf{k}) = 1$, which ignores the query. Figure 10.2 plots them. Each can be widened or narrowed by dividing the distance by a *bandwidth* σ. The same kernels estimate densities, where they are known as *Parzen windows* (Parzen 1957).

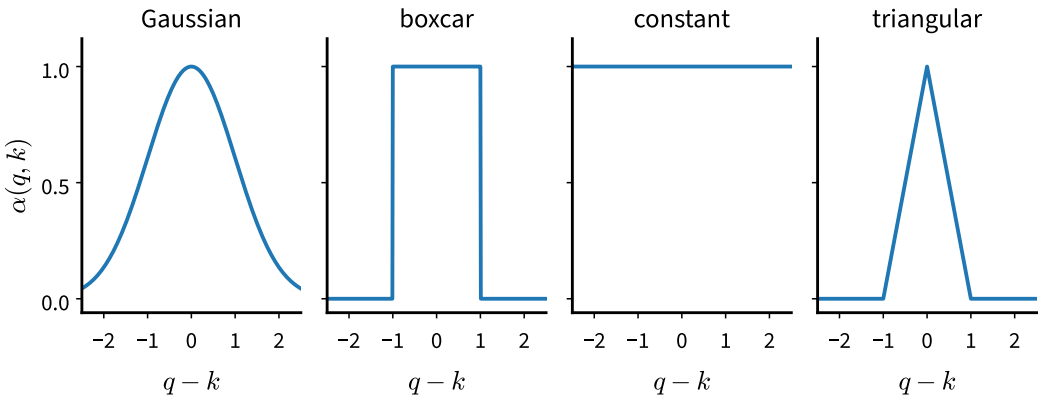


Figure 10.2: Four similarity kernels as functions of the query–key distance. Each can be rescaled by a bandwidth.

Normalizing a kernel over the keys, as in Equation 10.2 but without the exponential because kernels are already nonnegative, gives the *Nadaraya–Watson estimator*

$$f(\mathbf{q}) = \sum_i \mathbf{v}_i \frac{\alpha(\mathbf{q}, \mathbf{k}_i)}{\sum_j \alpha(\mathbf{q}, \mathbf{k}_j)}. \tag{10.4}$$

For scalar regression with observations (x_i, y_i), the keys are the training inputs x_i, the values are the labels y_i, and the query is the input at which we want a prediction. The estimate is a weighted average of the labels of the training inputs near the query, which is attention pooling term for term. It needs no training. If the bandwidth shrinks at a suitable rate as data accumulate, the estimate converges to the best possible predictor (Mack and Silverman 1982).

10.1.2.2 Nadaraya–Watson Regression

A one-dimensional regression problem is small enough to plot both the estimate and every weight behind it, so it shows what the bandwidth does before any learning is involved. We draw 40 training inputs uniformly from $[0, 5)$, label them by $y = 2\sin(x) + x + \epsilon$ with standard Gaussian noise ϵ, and keep a grid of noise-free points for comparison.

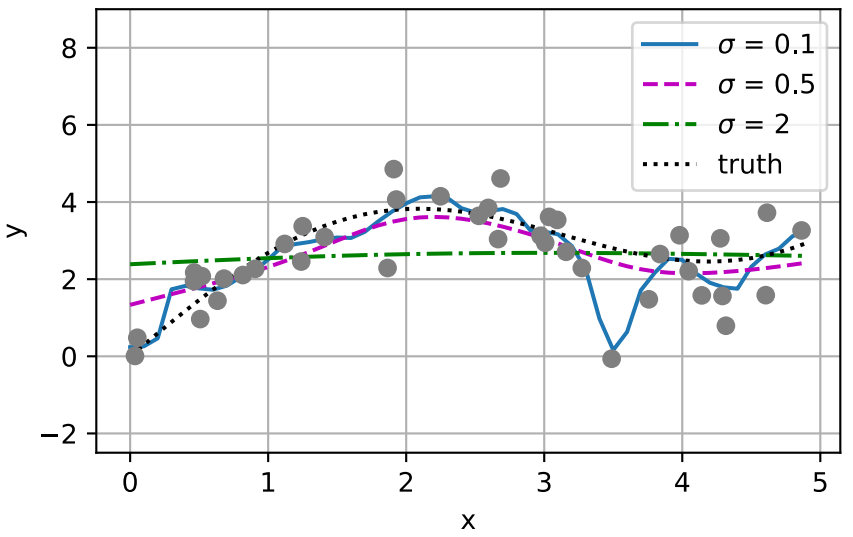
```
n = 40
x_train, _ = torch.sort(torch.rand(n) * 5)
y_train = 2 * torch.sin(x_train) + x_train + torch.randn(n)
x_val = torch.arange(0, 5, 0.1)
y_val = 2 * torch.sin(x_val) + x_val
```

The estimator computes all query–key distances, applies a Gaussian kernel of bandwidth σ, normalizes over the keys, and takes the weighted sum of the labels. The normalized kernel matrix is the matrix of attention weights, with one row per query, so we return it as well.

```
def nadaraya_watson(x_train, y_train, x_val, sigma):
    dists = x_val[:, None] - x_train[None, :]  # (queries, keys)
    k = d2l.exp(-dists**2 / (2 * sigma**2))
    attention_w = k / k.sum(1, keepdims=True)  # Normalize over the keys
    return attention_w @ y_train, attention_w
```

How much does the bandwidth matter? We compare three bandwidths, a factor of four or five apart, against the noise-free function; the dots are the training data.

```
sigmas = (0.1, 0.5, 2.0)
estimates = [nadaraya_watson(x_train, y_train, x_val, s)[0] for s in sigmas]
d2l.plot([x_val] * 4 + [x_train], estimates + [y_val, y_train], 'x', 'y',
         legend=[rf'$\sigma$ = {s:g}' for s in sigmas] + ['truth'],
         fmts=('-', 'm--', 'g-.', 'k:', 'C7o'), ylim=(-2.5, 9),
         figsize=(5, 3))
```



The bandwidth decides the fit. With $\sigma = 0.1$ the estimate chases the noise of individual observations. With $\sigma = 2$ it averages over most of the data and flattens toward the global mean, missing both the bump and the dip of the curve. The middle bandwidth, $\sigma = 0.5$, follows the noise-free function closely except near the ends of the interval, where all its neighbors lie on one side of the query and pull the estimate toward them. We keep the

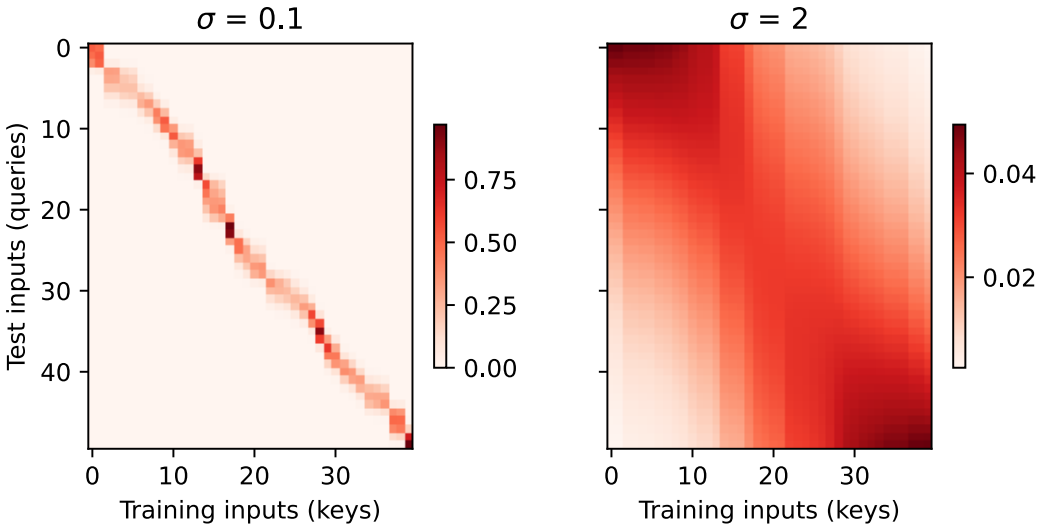
kernel Gaussian and vary only σ; the other shapes of Figure 10.2 can be rescaled in the same way.

10.1.3 Attention Weights as a Diagnostic

Because the weights are nonnegative and sum to one, a row of them reads as a distribution over keys. It tells us which stored pairs a query drew on. Plotting the weight matrix over (query, key) pairs is therefore the standard first look at any attention layer, and we will do it throughout the chapter with `d2l.show_heatmaps`. The helper takes an array of shape (rows of panels, columns of panels, queries, keys) and draws a grid of heatmaps.

The narrow and the wide kernel put weights of very different sizes on their keys, so we give each map its own color scale (`cbar='each'`). The queries are the test inputs and the keys are the training inputs, both in increasing order.

```
w_narrow, w_wide = (nadaraya_watson(x_train, y_train, x_val, s)[1]
                    for s in (0.1, 2.0))
d2l.show_heatmaps(d2l.stack([w_narrow, w_wide])[None],
                  xlabel='Training inputs (keys)',
                  ylabel='Test inputs (queries)',
                  titles=[r'$\sigma$ = 0.1', r'$\sigma$ = 2'],
                  figsize=(7, 3), cbar='each')
```



With the narrow kernel, each query puts nearly all of its weight on the few keys nearest to it, so the map is a thin diagonal band and the estimate copies a few noisy labels. The wide kernel spreads its weight out: its largest weight is a few hundredths, spread over a broad band of keys, so neighboring queries see nearly the same average. In both maps the diagonal appears only because queries and keys are sorted inputs on the same axis. In a trained network, such a heatmap is a hypothesis about what the layer does, not a proof. A large weight on a key whose value the next layer ignores explains nothing. Section 10.6 returns to this question with tools that test such hypotheses.

A heatmap also shows what Nadaraya–Watson regression cannot do. Its kernel fixes the

shape, the bandwidth, and the space in which distances are measured, so every query uses the same prescribed notion of similarity. A query cannot ask for the key three positions back, or for the key with the same grammatical role, because neither is a matter of distance in the input space. A learned attention layer instead computes queries and keys from the data with trainable maps, so that the weights they induce serve the task. The next section develops the scoring function that makes this practical.

10.1.4 Summary

Attention pooling Equation 10.1 answers a query with a weighted sum of stored values, where each weight measures how well the query matches the value's key. It reads every stored pair directly, so it needs no fixed-size summary of its input, and its parameters do not depend on how many pairs there are. The softmax turns any scoring function into nonnegative weights that sum to one; exact lookup and average pooling are the two extremes. Nadaraya–Watson regression is attention with a fixed kernel, and its bandwidth trades noise against bias. Heatmaps of the weights show which keys each query uses, and learned attention replaces the fixed kernel by learned queries and keys.

10.1.5 Exercises

1. **An average of two values.** Let the keys $\mathbf{k}_1, \dots, \mathbf{k}_m$ be orthonormal and let the score be $a(\mathbf{q}, \mathbf{k}) = \mathbf{q}^\top \mathbf{k}$ with the weights of Equation 10.2.
    1. For the query $\mathbf{q} = c\,(\mathbf{k}_a + \mathbf{k}_b)$ with $c > 0$, compute all m weights and show that the output tends to $\frac{1}{2}(\mathbf{v}_a + \mathbf{v}_b)$ as $c \to \infty$.
    2. How large must c be for the two selected keys to receive at least 99% of the total weight when $m = 10$?
    3. Now let the keys be orthogonal with $\|\mathbf{k}_b\| = 2\,\|\mathbf{k}_a\|$. Find a query that again averages $\mathbf{v}_a$ and $\mathbf{v}_b$, and state what $\mathbf{q} = c\,(\mathbf{k}_a + \mathbf{k}_b)$ returns instead as $c \to \infty$.

    Adapted from Stanford CS224N, Assignment 4, Problem 1(b).

2. code **Effective number of keys.** For the weights $\alpha_i(q)$ of one query, define the effective number of keys $n_{\text{eff}}(q) = 1/\sum_i \alpha_i(q)^2$.
    1. Let the values be $y_i = g(x_i) + \epsilon_i$ with independent noise of variance s^2, and hold the keys fixed. Show that the variance of the estimate Equation 10.4 at q is $s^2/n_{\text{eff}}(q)$.
    2. Compute n_{eff} at every query in `x_val` from the weights that `nadaraya_watson` returns, for each bandwidth in `sigmas`, and plot the three curves. Then redraw the noise in `y_train` 200 times with `x_train` fixed, and compare the standard deviation of the estimate at each query with $1/\sqrt{n_{\text{eff}}(q)}$.

3. code **Queries beyond the data.** The estimate Equation 10.4 is a convex combination of the values, and the inputs `x_train` lie in the interval $[0, 5)$.
    1. For distinct scalar keys and a Gaussian kernel of fixed bandwidth, find the limit of the estimate as $q \to \infty$. Find the same limit for the constant kernel.
    2. Evaluate `nadaraya_watson` with $\sigma = 0.5$ at the queries $5, 5.5, \dots, 10$ and plot the estimate against $y = 2\sin(x) + x$. Explain why no nonnegative kernel and no bandwidth let the estimate follow the trend of the truth.
    3. With $\sigma = 0.5$, `nadaraya_watson` returns NaN at the query $q = 15$. Explain why in single precision, and compute the weights instead as the softmax Equation 10.2

of the scores $-(q - x_i)^2/(2\sigma^2)$, so that the estimate is finite at every query and every bandwidth.

4. code **A learned bandwidth.** Treat σ in `nadaraya_watson` as a parameter and fit it by gradient descent on `x_train` and `y_train`.
   1. Before running, predict the minimizer of the training error $\sum_i (f(x_i) - y_i)^2$, given that y_i enters the computation of $f(x_i)$ through its own kernel weight.
   2. Run the optimization from $\sigma = 1$, report where σ goes, and compare the fitted curve with the estimates for the bandwidths in `sigmas`.
   3. Exclude (x_i, y_i) from the estimate at x_i by zeroing the diagonal of the kernel matrix and refit. Compare the learned σ with $\sigma = 0.5$, the best of the bandwidths in `sigmas`, and compute the mean squared error against `y_val` for both.

5. code **The gradient of attention.** ▨ Suppose that $a(\mathbf{q}, \mathbf{k}_i) = \mathbf{q}^\top \mathbf{k}_i$ and that $\mathbf{k}_i = \mathbf{v}_i$ for $i = 1, \dots, m$. Denote by $p(\mathbf{k}_i; \mathbf{q})$ the distribution over keys defined by the softmax normalization Equation 10.2.
   1. Prove that
   $$\nabla_{\mathbf{q}} \operatorname{Attention}(\mathbf{q}, \mathcal{D}) = \operatorname{Cov}_{p(\mathbf{k}_i;\mathbf{q})}[\mathbf{k}_i].$$
   2. Conclude that this Jacobian is positive semidefinite, and determine for which databases it is the zero matrix.
   3. Let $\mathbf{q} = c\,\mathbf{u}$ for a fixed direction $\mathbf{u}$ for which exactly one key attains the largest score $\mathbf{u}^\top \mathbf{k}_i$. Find the limit of the Jacobian as $c \to \infty$, and state what it implies for training the query by gradient descent.
   4. Verify the identity and the limit with automatic differentiation on a random database with $m = 8$ and $d = 4$.

6. code **Soft lookup as retrieval.** ▨ Build a database of m keys drawn uniformly from the unit sphere in $\mathbb{R}^{16}$, with value $\mathbf{v}_i$ the one-hot indicator of index i, and weight them with the Gaussian kernel of bandwidth σ normalized as in Equation 10.4. A query $\mathbf{q} = \mathbf{k}_i + \epsilon$ with $\epsilon \sim \mathcal{N}(\mathbf{0}, \tau^2 \mathbf{I})$ is retrieved correctly when the largest entry of the output is entry i.
   1. Show that the retrieved index is the index of the nearest key for every $\sigma > 0$, so the bandwidth does not affect which index is retrieved.
   2. Without noise, the weight on the correct key is
   $$\alpha_i = \frac{1}{1 + \sum_{j \neq i} \exp\big(-\|\mathbf{k}_i - \mathbf{k}_j\|^2 / 2\sigma^2\big)}. \tag{10.5}$$
   Approximate $\mathbf{k}_i^\top \mathbf{k}_j \approx 0$ for random unit vectors, so that $\|\mathbf{k}_i - \mathbf{k}_j\|^2 \approx 2$, and derive the bandwidth at which α_i falls to 0.9 for $m = 20$ and for $m = 200$. How must σ shrink as m grows?
   3. Measure α_i at these two bandwidths, averaged over random databases, and explain the direction of its discrepancy from 0.9.
   4. Sweep τ from 0 to 1 at $m = 20$ and $m = 200$, with the bandwidth from (b) for each m, and plot the retrieval accuracy against τ. Compute the weights as the softmax Equation 10.2 of the scores $-\|\mathbf{q} - \mathbf{k}_j\|^2 / 2\sigma^2$. Explain why a larger m lowers the accuracy at fixed τ although the distance from the query to the correct key is unchanged.

7. [extended] **A learned metric.** ▯ Kernel regression with a learned distance (Weinberger and Tesauro 2007) replaces the fixed similarity of Equation 10.4 by a learned one. Draw $n = 200$ inputs $\mathbf{x}_i$ uniformly from $[0, 5]^5$ and labels $y_i = 2\sin(x_{i1}) + x_{i1} + \epsilon_i$ with standard Gaussian noise, so that only the first coordinate matters. Score a query against a key by

$$a(\mathbf{q}, \mathbf{k}) = -\tfrac{1}{2}\|\mathbf{W}(\mathbf{q} - \mathbf{k})\|^2, \qquad \mathbf{W} = \mathrm{diag}(w_1, \dots, w_5), \tag{10.6}$$

normalize as in Equation 10.2, and fit $\mathbf{W}$ to the leave-one-out error, in which the estimate at $\mathbf{x}_i$ excludes the pair $(\mathbf{x}_i, y_i)$. Measure the test error on 1000 fresh inputs against the noise-free function.

   1. Fit the isotropic kernel $w_1 = \cdots = w_5 = 1/\sigma$ by a grid search over σ.
   2. Starting from the isotropic solution, fit the five scales by gradient descent. Compare the learned scales with the relevance of each coordinate, and the test error with that of the isotropic kernel and of an oracle that uses only the first coordinate at its best bandwidth.
   3. Compare the leave-one-out errors of the learned and the oracle metrics with their test errors, and repeat the comparison at $n = 1000$. What does the result say about learning a similarity from a finite sample?
   4. A full matrix $\mathbf{W} \in \mathbb{R}^{5\times 5}$ maps queries and keys by the same linear map, whereas learned attention maps them by separate matrices $\mathbf{W}_q$ and $\mathbf{W}_k$. State two properties of the score a that the shared map guarantees and separate maps lose.

10.2 Attention Scoring and Masking

In Section 10.1.2 the attention weights came from a kernel chosen by hand. They follow distance in a fixed input space, and no amount of data changes what counts as similar. In a sentence, however, a verb looking for its subject wants the word that plays that grammatical role, which need not be the word whose representation is closest to its own. Turning soft lookup into a layer of a trained network raises two questions: which score should the network learn, and which query–key pairs should count at all?

The score must be shaped by learned representations, and it must be cheap, because it is evaluated for every pair: one attention head of GPT-2 small, reading a context of 1,024 tokens, scores $1{,}024^2 = 1{,}048{,}576$ pairs. Some of those pairs must not count. Sequences of different lengths share a minibatch only after padding, and no query should attend to the padding. A language model trained to predict each next token processes all positions at once, and the query at position t must not see the tokens after it.

This section answers both questions. The score is the dot product of query and key, divided by $\sqrt{d}$ so that its scale does not grow with the width d of the vectors. Exclusions are boolean masks applied before the softmax, and several of them combine by logical AND. With both in place, attention over a whole minibatch is two batched matrix multiplications and a softmax, the `DotProductAttention` layer that every later model in this book builds on. The section ends with the translation model in which learned attention first appeared.

10.2.1 Dot-Product Attention

We look for a score that learned queries and keys can shape and that costs one short computation per pair. The Gaussian kernel of Section 10.1.2 already contains one. Expanding its exponent gives the score

$$a(\mathbf{q}, \mathbf{k}_i) = -\frac{1}{2}\|\mathbf{q} - \mathbf{k}_i\|^2 = \mathbf{q}^\top \mathbf{k}_i - \frac{1}{2}\|\mathbf{k}_i\|^2 - \frac{1}{2}\|\mathbf{q}\|^2. \tag{10.7}$$

The last term depends on the query only. It is the same for all keys, and the softmax normalization Equation 10.2 removes it. If all keys have the same norm, the middle term drops out in the same way, and the Gaussian kernel and the dot product give identical weights. In general the norms differ, and keeping only $\mathbf{q}^\top \mathbf{k}_i$ is a modeling decision rather than an approximation: the dot product becomes the compatibility function in its own right. Learned query and key representations then decide what compatibility means, and the Gaussian kernel remains the special case of keys with equal norms.

One adjustment controls the size of the scores. Suppose that the d coordinates of the query and of the key are independent random variables with mean zero and variance one. Their dot product then has mean zero and variance d; for the GPT-2 head, with $d = 64$, a typical score is about ± 8. Dividing by $\sqrt{d}$ restores unit variance and gives the *scaled dot-product* score of the Transformer (Vaswani et al. 2017):

$$a(\mathbf{q}, \mathbf{k}_i) = \mathbf{q}^\top \mathbf{k}_i / \sqrt{d}. \tag{10.8}$$

The attention weights are, as always, the softmax of the scores:

$$\alpha(\mathbf{q}, \mathbf{k}_i) = \text{softmax}(a(\mathbf{q}, \mathbf{k}_i)) = \frac{\exp(\mathbf{q}^\top \mathbf{k}_i / \sqrt{d})}{\sum_{j=1}^{m} \exp(\mathbf{q}^\top \mathbf{k}_j / \sqrt{d})}. \tag{10.9}$$

10.2.1.1 Score Variance and the $1/\sqrt{d}$ Factor

Learned queries and keys are neither independent nor of unit variance, so the calculation above describes a network at initialization. That is when it matters. If one score exceeds the others by a wide margin, its weight approaches 1, the others approach 0, and the softmax *saturates*. The gradient that reaches the scores shrinks with it, because the Jacobian of the softmax is

$$\frac{\partial \alpha}{\partial \mathbf{a}} = \text{diag}(\alpha) - \alpha\alpha^\top, \tag{10.10}$$

which tends to the zero matrix as α approaches a one-hot vector. Row i of the Jacobian carries the factor α_i, so the gradient of the loss with respect to score i is proportional to the weight of key i. In a saturated row, the keys with negligible weight receive almost no gradient, and the query learns only about the few keys that already carry weight. A key that loses at the start has little chance to catch up. Only this route saturates. Gradients still reach the values, the output projection, and any residual connection. For finite scores the Jacobian is never exactly zero, since the all-ones vector alone spans its null space (an exercise below proves this), but it can come arbitrarily close.

We measure both effects. We draw random queries and keys with independent standard normal entries, compute the attention weights of each query over 64 keys, and record the entropy of the weights as the dimension d grows, with and without the factor $1/\sqrt{d}$. For reference, the uniform distribution over 64 keys has entropy $\log 64 \approx 4.2$ nats, and a one-hot distribution has entropy 0.

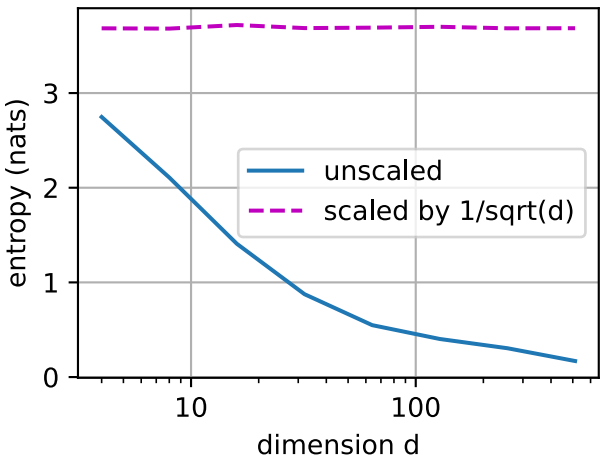
```
def softmax_weights(d, scale, num_keys=64, num_queries=256):
    """Log-weights of random queries over random keys, scores scaled."""
    q = torch.randn(num_queries, d)
    keys = torch.randn(num_queries, num_keys, d)
    scores = (keys @ q[..., None]).squeeze(-1) * scale
    return F.log_softmax(scores, dim=-1)

def entropy(log_alpha):
    return -(log_alpha.exp() * log_alpha).sum(-1).mean().item()

def jacobian_norm(log_alpha):
    """Mean Frobenius norm of the Jacobian diag(alpha) - alpha alpha^T."""
    alpha = log_alpha.exp()
    J = torch.diag_embed(alpha) - alpha[..., :, None] * alpha[..., None, :]
    return torch.linalg.matrix_norm(J).mean().item()
```

The entropy uses the log-weights directly, so a weight that underflows to zero contributes exactly zero rather than $0 \cdot \log 0$.

```
ds = [2**k for k in range(2, 10)]
entropies = [[entropy(softmax_weights(d, 1.0)) for d in ds],
             [entropy(softmax_weights(d, 1 / math.sqrt(d))) for d in ds]]
d2l.plot(ds, entropies, 'dimension d', 'entropy (nats)',
         legend=['unscaled', 'scaled by 1/sqrt(d)'], xscale='log')
```



With scaling, the entropy stays near 3.7 nats over two orders of magnitude of d: the weights are equally sharp at every width. Without scaling it collapses as d grows, to about 0.2 nats at $d = 512$, a nearly one-hot weighting that comes from the length of the vectors alone. Unscaled, the largest of the 64 weights takes about 80% of the total at $d = 64$ and about 90% at $d = 512$. The norm of the Jacobian shows what this does to the gradient that reaches the scores:

```
for d in (4, 64, 512):
    unscaled = jacobian_norm(softmax_weights(d, 1.0))
    scaled = jacobian_norm(softmax_weights(d, 1 / math.sqrt(d)))
    print(f'd = {d:3d}: Jacobian norm {unscaled:.3f} (unscaled), '
          f'{scaled:.3f} (scaled)')
```

```
d =   4: Jacobian norm 0.254 (unscaled), 0.177 (scaled)
d =  64: Jacobian norm 0.200 (unscaled), 0.183 (scaled)
d = 512: Jacobian norm 0.092 (unscaled), 0.185 (scaled)
```

With scaling, the norm is the same at every dimension. Without it, the norm first *rises*: at small d the unscaled weights concentrate on a few keys, and the Jacobian is largest when two keys share the weight (an exercise below finds this maximum). It crosses the scaled value near $d = 128$, falls to about half of it at $d = 512$, and keeps falling. A model with unscaled scores would therefore start training with nearly one-hot attention, and the gradient that reaches its scores would flow almost entirely through the few keys that each query already favors. The factor $1/\sqrt{d}$ removes this dependence on the dimension. That is why it is part of the definition Equation 10.8 rather than a hyperparameter to tune.

10.2.2 Masking

So far every query attends to every key. Two situations require exclusions. The first is *padding*. Sequences of different lengths share a minibatch only after they are padded with a dummy token to a common length (here `<blank>`):

```
Dive  into  Deep    Learning
Learn to    code    <blank>
Hello world <blank> <blank>
```

The padding carries no meaning, and no query should give it weight. The second is *causality*. A language model trained to predict the next token computes its outputs for all positions of a sequence at once, and the query at position t must not see the keys at positions after t. Otherwise it could copy the very token it is asked to predict.

Both are described by one object, a boolean *mask* with one entry per query–key pair that is true where the query may attend to the key. The true entries of a row need not form a prefix, so the same object also describes the attention windows and document boundaries we meet later.

10.2.2.1 The Masked Softmax Operation

The *masked softmax* gives zero weight to the excluded pairs. It does not skip them, which would leave rows of different lengths. Instead it overwrites the excluded scores with a very negative number before the softmax, so that their exponentials vanish, and keeps the regular array shapes that batched kernels need.

The number has to be chosen with the precision in mind. Older code used a constant such as -10^6. That works in single precision. Float16, a format used for training today, cannot represent numbers beyond about 6.5×10^4, and depending on the framework -10^6 either overflows to $-\infty$ or is refused with an error. Literal $-\infty$ also masks correctly, until some query has no valid key at all. Every score of its row is then $-\infty$, the softmax subtracts the row maximum and computes $-\infty - (-\infty)$, and the row becomes NaN. The NaN spreads to

everything computed from it. We therefore overwrite with the most negative *finite* value of the dtype (`finfo(dtype).min`). After the softmax subtracts the row maximum, its exponential underflows to exactly zero at any precision, and no NaN can arise.

One case remains. A query whose keys are all excluded would now receive the uniform average of invalid values, because all its scores are equal. `masked_softmax` sets such rows to zero. A model should still ensure that every query keeps at least one valid key.

```
def masked_softmax(X, mask=None):
    """Softmax over the last axis with zero weight where mask is False."""
    if mask is None:
        return F.softmax(X, dim=-1)
    # Most negative finite value: zero weight at any precision, no NaN
    X = X.masked_fill(~mask, torch.finfo(X.dtype).min)
    weights = F.softmax(X, dim=-1)
    # A query without any valid key gets zero weights, not a uniform row
    return torch.where(mask.any(-1, keepdim=True), weights, 0.0)
```

The mask need only *broadcast* against the scores, so a mask that is the same for every query, or for every sequence, is stored once. A padding mask is of this kind: key j of a sequence is valid when j is less than the sequence's valid length. `padding_mask` turns a vector of valid lengths, one per sequence, into a mask of shape (batch, 1, keys), which the softmax broadcasts over the queries. A matrix of lengths, one per query, gives a mask of shape (batch, queries, keys).

```
def padding_mask(valid_lens, num_keys):
    """Boolean mask that is True for the first valid_lens keys."""
    # valid_lens of shape (batch,) gives (batch, 1, keys); per-query
    # lengths of shape (batch, queries) give (batch, queries, keys)
    if valid_lens.dim() == 1:
        valid_lens = valid_lens[:, None]
    keys = torch.arange(num_keys, device=valid_lens.device)
    return keys < valid_lens[..., None]
```

We try both on a minibatch of two sequences with two queries and four keys each, and with valid lengths 2 and 3. The weights beyond the valid length should be zero, and each row should still sum to 1.

```
masked_softmax(torch.rand(2, 2, 4), padding_mask(torch.tensor([2, 3]), 4))
```

```
tensor([[[0.4920, 0.5080, 0.0000, 0.0000],
         [0.4500, 0.5500, 0.0000, 0.0000]],

        [[0.3732, 0.2170, 0.4098, 0.0000],
         [0.4212, 0.3096, 0.2692, 0.0000]]])
```

With one valid length per query, each row is cut at its own length:

```
masked_softmax(torch.rand(2, 2, 4),
               padding_mask(torch.tensor([[1, 3], [2, 4]]), 4))
```

```
tensor([[[1.0000, 0.0000, 0.0000, 0.0000],
         [0.4348, 0.3049, 0.2604, 0.0000]],

        [[0.5471, 0.4529, 0.0000, 0.0000],
         [0.2523, 0.2052, 0.2362, 0.3063]]])
```

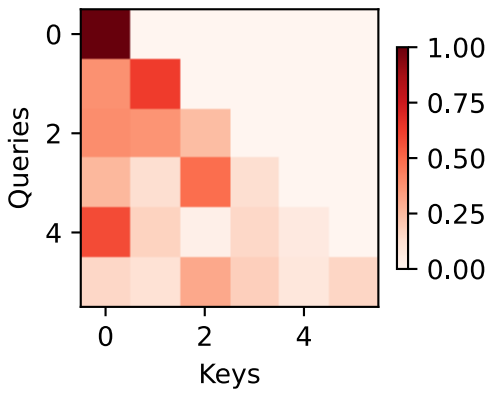
10.2.2.2 Causal Masking

In a causal mask, the query at position i may see the keys at positions $j \leq i$. The mask is a lower triangle, and in a sequence of length n it excludes $n(n-1)/2$ of the n^2 pairs, almost half of them. For one head of GPT-2 small reading $n = 1{,}024$ tokens, that is $523{,}776$ of the $1{,}048{,}576$ scores. `causal_mask` also covers the case of fewer queries than keys, which arises when new tokens attend to the keys of earlier tokens kept in a cache (Section 11.3). The queries are then the *last* positions of the sequence, and query i may see key j when $j \leq i + (\text{keys} - \text{queries})$.

```
def causal_mask(num_queries, num_keys=None, device=None):
    """Boolean (queries, keys) mask, True where a query may see a key."""
    # Queries are the last positions, as when new tokens attend to a
    # cache: query i sees key j if j <= i + num_keys - num_queries
    num_keys = num_queries if num_keys is None else num_keys
    i = torch.arange(num_queries, device=device)[:, None]
    j = torch.arange(num_keys, device=device)[None, :]
    return j <= i + (num_keys - num_queries)
```

On six random scores, the masked weights form a lower triangle:

```
scores = torch.randn(1, 6, 6)
d2l.show_heatmaps(masked_softmax(scores, causal_mask(6))[None],
                  xlabel='Keys', ylabel='Queries')
```



The first query attends only to itself, and each later query spreads its weight over itself and its predecessors. The triangle lets a model train on all positions of a sequence in parallel without showing any position its future. Every decoder in the following chapters uses it.

10.2.2.3 Composing Masks

Each requirement excludes some pairs. Padding excludes the keys beyond a sequence's length, causality the keys after the query, and the attention windows of Section 10.5 the keys too far from it. A pair is allowed only if every requirement allows it, so masks compose by logical AND. Broadcasting keeps the composite small. A padding mask of shape (batch, 1, keys) and a causal mask of shape (queries, keys) combine into a mask of shape (batch, queries, keys), and neither is copied per example or per query before the AND (Figure 10.3).

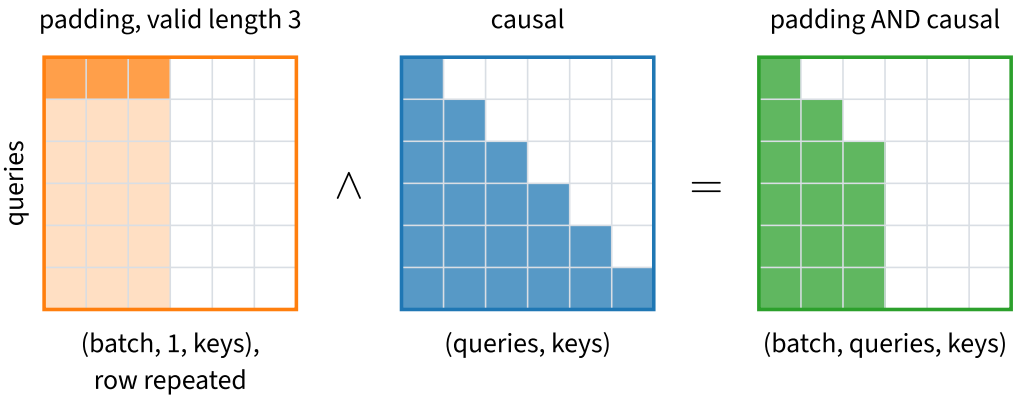
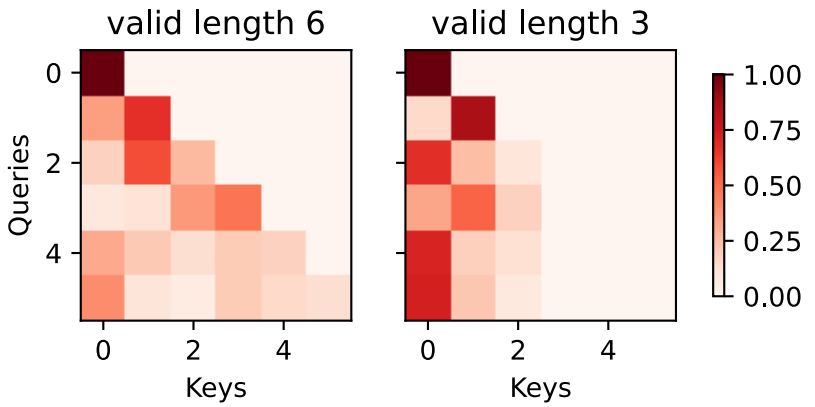


Figure 10.3: Masks compose by logical AND. For a sequence of length 6 with valid length 3, the padding mask stores one row of keys, which broadcasting repeats for every query (faint); the causal mask is a lower triangle; their AND allows a pair only if both do. Rows are queries, columns are keys, and filled entries are allowed.

Here are the weights for a minibatch of two sequences of length 6 with valid lengths 6 and 3:

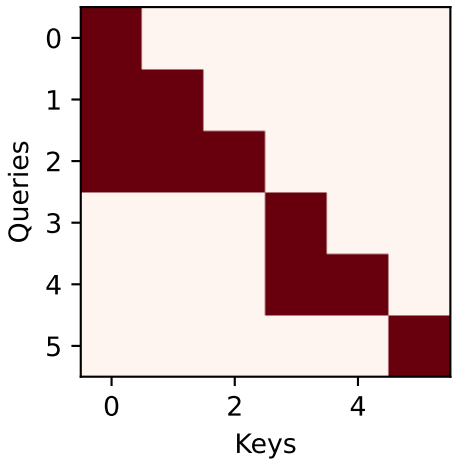
```
n = 6
padding = padding_mask(torch.tensor([6, 3]), n)  # (batch, 1, key)
causal = causal_mask(n)                          # (query, key)
mask = padding & causal                          # (batch, query, key)
weights = masked_softmax(torch.randn(2, n, n), mask)
d2l.show_heatmaps(weights[None], xlabel='Keys', ylabel='Queries',
                  titles=['valid length 6', 'valid length 3'],
                  figsize=(5, 2.5))
```



The first sequence shows the plain causal triangle. The second has valid length 3, so every row stops at the third key. Composition makes empty rows more likely. Masks that each leave every query a key can still have an empty intersection, so the guarantee of a valid key must hold for the composite mask.

The same algebra handles *packed sequences*. Padding many short documents to a common length wastes computation on the padding, so training pipelines concatenate documents into one row instead. A mask that allows a pair only within one document, ANDed with the causal mask, keeps the documents apart. Here a row of length 6 holds three documents of lengths 3, 2, and 1:

```
doc = torch.tensor([0, 0, 0, 1, 1, 2])  # Document of each position
same_doc = doc[:, None] == doc[None, :]
packed = same_doc & causal_mask(6)
d2l.show_heatmaps(packed.float()[None, None], xlabel='Keys',
                  ylabel='Queries', cbar=False)
```



The composite mask is block diagonal, with one causal triangle per document: each position attends only to the earlier positions of its own document.

10.2.3 Batched Attention

10.2.3.1 Batch Matrix Multiplication

Attention is computed on minibatches of queries, keys, and values, so we need to multiply batches of matrices with each other. Consider two batches of n matrices each,

$$\mathbf{Q} = [\mathbf{Q}_1, \mathbf{Q}_2, \ldots, \mathbf{Q}_n] \in \mathbb{R}^{n \times a \times b}, \qquad \mathbf{K} = [\mathbf{K}_1, \mathbf{K}_2, \ldots, \mathbf{K}_n] \in \mathbb{R}^{n \times b \times c}.$$

The *batch matrix multiplication* (BMM) computes one matrix product per batch element,

$$\mathrm{BMM}(\mathbf{Q}, \mathbf{K}) = [\mathbf{Q}_1\mathbf{K}_1, \mathbf{Q}_2\mathbf{K}_2, \ldots, \mathbf{Q}_n\mathbf{K}_n] \in \mathbb{R}^{n \times a \times c}. \tag{10.11}$$

The `@` operator computes it. It batches over any number of leading axes, which the attention layers of the next sections use for their extra axis of heads.

```
Q = torch.ones((2, 3, 4))
K = torch.ones((2, 4, 6))
d2l.check_shape(Q @ K, (2, 3, 6))
```

`torch.bmm(Q, K)` computes the same product for arrays with exactly one batch axis. The `@` operator, short for `torch.matmul`, also accepts any number of leading axes and broadcasts them.

10.2.3.2 The DotProductAttention Class

Scaled dot-product attention can now be written for many queries at once. For n queries and m key–value pairs, with queries and keys of width d and values of width d_v, stack the queries into $\mathbf{Q} \in \mathbb{R}^{n \times d}$, the keys into $\mathbf{K} \in \mathbb{R}^{m \times d}$, and the values into $\mathbf{V} \in \mathbb{R}^{m \times d_v}$. The whole computation is two matrix products and a softmax (Figure 10.4):

$$\mathrm{softmax}\left(\frac{\mathbf{Q}\mathbf{K}^\top}{\sqrt{d}}\right)\mathbf{V} \in \mathbb{R}^{n \times d_v}. \tag{10.12}$$

The softmax normalizes each row of the score matrix, so row i of the output is the attention Equation 10.1 of query i. The cost is visible in the shapes. For the GPT-2 head at a context of $n = m = 1{,}024$ with $d = d_v = 64$, the score matrix holds $1{,}048{,}576$ entries, 4 MiB in single precision, and the two products take $2 \cdot 2n^2 d \approx 0.27$ billion floating-point operations. Both numbers grow with the square of the context length, which is the subject of Section 10.5.

Queries and keys must have the same width d. That is no restriction in practice, because the queries and keys of a network are computed from its inputs by learned linear maps, which can map to any common width (Section 10.3). On a minibatch, Equation 10.12 uses the batch matrix multiplication Equation 10.11 twice. The class applies dropout to the attention weights as a regularizer and keeps the weights of its last call in `attention_weights` for inspection.

```
class DotProductAttention(nn.Module):
    """Scaled dot-product attention over the last two axes."""
```

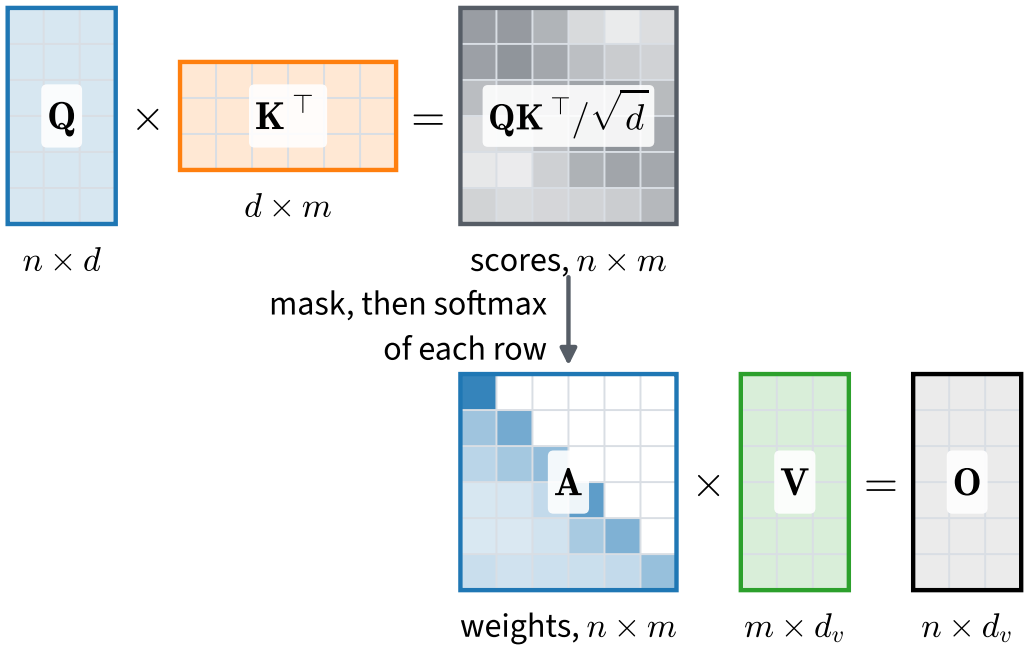


Figure 10.4: Shapes in scaled dot-product attention. $\mathbf{QK}^\top/\sqrt{d}$ holds the scores of all $n \times m$ query–key pairs; masking and a softmax over each row turn them into the weights $\mathbf{A}$ (here with a causal mask, which leaves the upper triangle empty); multiplying by $\mathbf{V}$ gives one weighted sum of values per query.

```
    def __init__(self, dropout=0.0):
        super().__init__()
        self.dropout = nn.Dropout(dropout)
        self.attention_weights = None

    # Leading axes (batch, and heads if present) are carried along.
    # Shape of queries: (..., no. of queries, d)
    # Shape of keys: (..., no. of key-value pairs, d)
    # Shape of values: (..., no. of key-value pairs, value dimension)
    # mask: boolean, broadcastable to (..., no. of queries, no. of keys),
    # True where a query may attend to a key
    def forward(self, queries, keys, values, mask=None):
        d = queries.shape[-1]
        scores = queries @ keys.transpose(-2, -1) / math.sqrt(d)
        self.attention_weights = masked_softmax(scores, mask)
        return self.dropout(self.attention_weights) @ values
```

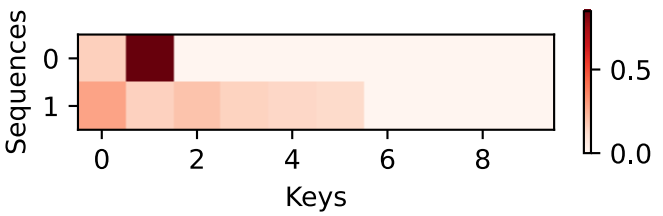
We test the class on a minibatch of two sequences, each with one query of width 2 and ten key–value pairs with values of width 4, and with valid lengths 2 and 6. Evaluation mode switches dropout off. The output has one row of width 4 per query.

```
queries = torch.randn(2, 1, 2)
keys = torch.randn(2, 10, 2)
values = torch.randn(2, 10, 4)
mask = padding_mask(torch.tensor([2, 6]), 10)

attention = DotProductAttention(dropout=0.5)
attention.eval()
d2l.check_shape(attention(queries, keys, values, mask), (2, 1, 4))
```

The stored weights confirm the mask. Each row holds the weights of one sequence's query, and they vanish beyond the second and the sixth key, respectively.

```
d2l.show_heatmaps(attention.attention_weights.reshape((1, 1, 2, 10)),
                  xlabel='Keys', ylabel='Sequences', figsize=(4, 1.6))
```



10.2.4 Attention as Learned Alignment

Learned attention first appeared in neural machine translation. Around 2014 the leading neural approach encoded a source sentence with an RNN into one fixed-size vector and decoded the translation from that vector with a second RNN; we build such an encoder–

decoder in Section 43.1. This is the bottleneck of Section 10.1: one vector had to represent the whole sentence, and translation quality fell as sentences grew longer.

Bahdanau et al. (2015) kept the two RNNs but changed what the decoder reads. At every output step, the current decoder state served as a query against all encoder states, which served as keys and values. The weighted sum, a fresh one for every output token, entered the next prediction. The paper's title called this "jointly learning to align and translate". The learned weights resembled the soft word alignments of classical statistical translation: mostly monotone along the diagonal, with departures where the two languages order words differently, as sketched in Figure 10.5. Nothing in the training objective asked for alignments. They emerged because they help to translate.

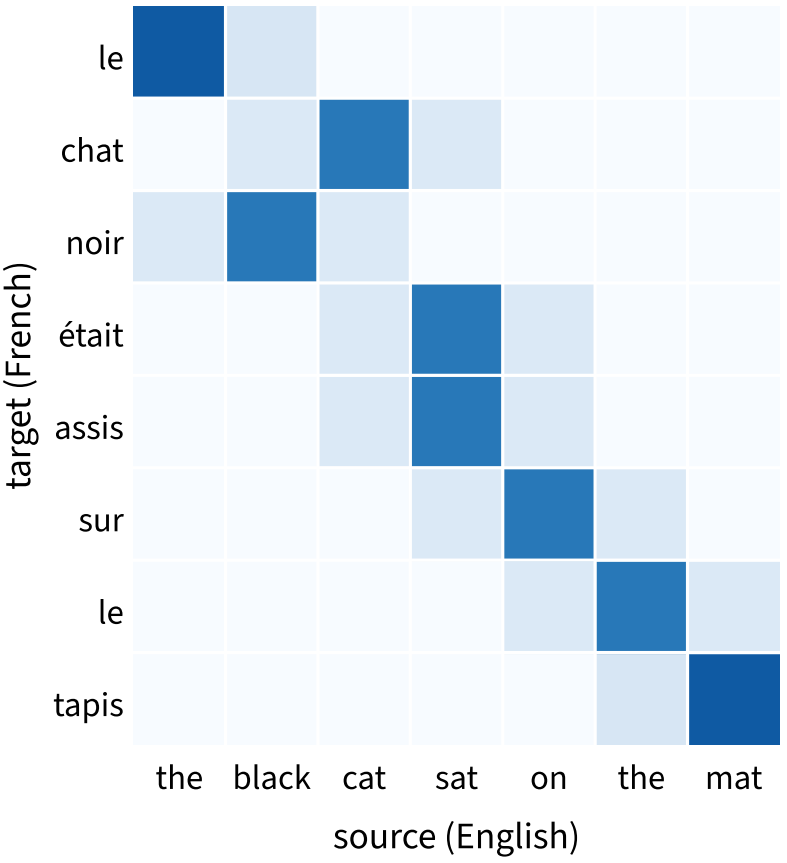


Figure 10.5: Soft alignment between an English sentence and its French translation, in the style of a learned attention map (darker cells indicate larger weight; schematic). The alignment is mostly monotone, but "black cat" maps to "chat noir" with the order reversed, and both "était" and "assis" draw on "sat".

One detail differed from the score we chose above. The decoder state and the encoder states had different widths, so instead of a dot product, Bahdanau et al. (2015) scored with a small network with one hidden layer, now called *additive attention*:

$$a(\mathbf{q}, \mathbf{k}) = \mathbf{w}_v^\top \tanh(\mathbf{W}_q\mathbf{q} + \mathbf{W}_k\mathbf{k}) \in \mathbb{R}. \tag{10.13}$$

Here $\mathbf{q} \in \mathbb{R}^{d_q}$ and $\mathbf{k} \in \mathbb{R}^{d_k}$ may have different widths, and the learned parameters are $\mathbf{W}_q \in \mathbb{R}^{r \times d_q}$, $\mathbf{W}_k \in \mathbb{R}^{r \times d_k}$, and $\mathbf{w}_v \in \mathbb{R}^r$ for a hidden width r. The two projections map queries and keys into a shared r-dimensional space, and $\mathbf{w}_v$ reads a score off their sum. We score three queries of width 20 against six keys of width 2, widths that no dot product could pair.

```
queries, keys = torch.randn(3, 20), torch.randn(6, 2)
r = 8  # Hidden width
W_q = torch.randn(r, 20) / math.sqrt(20)
W_k = torch.randn(r, 2) / math.sqrt(2)
w_v = torch.randn(r) / math.sqrt(r)
features = torch.tanh((queries @ W_q.T)[:, None, :]
                      + (keys @ W_k.T)[None, :, :])
scores = features @ w_v
d2l.check_shape(scores, (3, 6))
F.softmax(scores, dim=-1)
```

```
tensor([[0.2818, 0.1473, 0.1029, 0.2014, 0.0971, 0.1696],
        [0.2419, 0.1682, 0.0942, 0.2157, 0.1169, 0.1631],
        [0.2208, 0.1576, 0.1162, 0.2236, 0.1337, 0.1481]])
```

Each row is a distribution over the six keys. The intermediate array `features` has shape $3 \times 6 \times 8$, one hidden vector per query–key pair, and this array is why the dot product replaced the additive score. All dot products of queries and keys form one matrix multiplication, which accelerators execute fast, while the additive score needs an $n \times m \times r$ array of hidden activations. The dot product can still use a learned metric. Projecting queries and keys with learned matrices first leaves a score that runs at the speed of matrix multiplication. This is the form attention takes in the Transformer, whose authors dropped the recurrent networks altogether and kept attention as the only operation that relates positions (Vaswani et al. 2017).

We can now compute attention for a whole minibatch, with scores that learned representations can shape and with any pattern of exclusions. Each query, however, still forms a single weighted average, and nothing in the computation knows where in the sequence a key sits. Section 10.3 runs several lookups side by side, and Section 10.4 adds positions.

10.2.5 Summary

Scaled dot-product attention scores a query against a key by $\mathbf{q}^\top\mathbf{k}/\sqrt{d}$, which is the Gaussian kernel without the norm terms, and it scores all pairs of a minibatch with one batched matrix multiplication. The factor $1/\sqrt{d}$ keeps the score variance independent of the width, so the softmax does not saturate at initialization merely because the vectors are long. A boolean mask excludes padding, future positions, or any other pairs before the softmax. Masks compose by logical AND under broadcasting. The excluded scores are overwritten with the most negative finite value of the dtype, which is safe in reduced precision. `DotProductAttention` combines scoring, masking, dropout, and the weighted sum of values. Additive attention, the score of the first neural alignment model, gave way to the dot product because the dot product is a matrix multiplication.

10.2.6 Exercises

1. **Invariances of the score.** The Gaussian score Equation 10.7 and the scaled dot product Equation 10.8 both define attention weights through the softmax Equation 10.9.
   1. For each of the two scores, determine which of the following transformations leaves the weights unchanged: adding one vector $\mathbf{c}$ to every key; adding $\mathbf{c}$ to the query and to every key; multiplying the query and every key by one orthogonal matrix.
   2. Queries and keys are usually computed from inputs by affine maps, $\mathbf{q} = \mathbf{W}_q\mathbf{x}+\mathbf{b}_q$ and $\mathbf{k}_i = \mathbf{W}_k\mathbf{y}_i + \mathbf{b}_k$. Expand $\mathbf{q}^\top\mathbf{k}_i$ into four terms and show that the weights do not depend on $\mathbf{b}_k$. What is the gradient of the loss with respect to $\mathbf{b}_k$?
   3. Through which of the four terms does $\mathbf{b}_q$ affect the weights, and what preference over the keys can it express that the bias-free score cannot?
2. code **The softmax Jacobian.** `jacobian_norm` measures the Frobenius norm of the Jacobian Equation 10.10.
   1. Derive Equation 10.10 from Equation 10.9, show that for finite scores its null space is spanned by the all-ones vector, and check the formula against automatic differentiation on a random score vector.
   2. Compute the Frobenius norm when α is one-hot and when it is uniform over m keys, and evaluate the uniform case for $m = 64$. Show that the squared norm is close to $\sum_i \alpha_i^2$ when every weight is small, and use this to explain why the scaled norms printed for $d \in \{4, 64, 512\}$ exceed the uniform value.
   3. Compute the norm for weights spread uniformly over k of the m keys, find the k that maximizes it, and confirm by sampling random weight vectors that none exceeds this maximum. Use the result to explain why the unscaled norm printed for $d = 4$ exceeds the scaled one.
3. **Fully masked rows.** `masked_softmax` overwrites excluded scores with the most negative finite value of the dtype and then zeroes the rows of queries that have no valid key.
   1. Explain why, without the final zeroing, a fully masked row returns the uniform distribution, and why the same row returns NaN when the scores are overwritten with $-\infty$ instead. Trace both cases through the subtraction of the row maximum that a numerically stable softmax performs.
   2. In a self-attention layer, whose queries, keys, and values all come from the same sequence, let a query at a padding position have no valid key, as the composite masks of Section 10.2.2.3 can arrange, and let the excluded scores be overwritten with $-\infty$, so that the output at that position is NaN. That output is discarded, and in the next attention layer every valid query excludes the padding position as a key. Explain why the NaN nevertheless corrupts the outputs of all valid positions of that sequence in the next layer.
4. **Composite masks.** Section 10.2.2.3 combines masks by logical AND and warns that masks that are harmless alone can leave a query with no valid key. A causal sliding window of width w lets the query at position t attend to the keys at positions $t - w + 1, \dots, t$.
   1. For a sequence of length 6 with valid length 3, write the padding mask and the causal sliding window of width 2 as 6×6 boolean matrices. Check that each leaves every query at least one key, and find the queries whose row of the AND is empty.
   2. Show that an AND of masks has no empty row if every mask allows each query to attend to its own position. Which of the padding, causal, sliding-window, and packed-sequence masks satisfy this condition, and what does the answer say

about the empty rows you found?

3. Some implementations add a bias matrix with entries 0 for allowed pairs and $c < 0$ for excluded pairs, and compose masks by adding their biases. Take for c the most negative finite single-precision value, about -3.4×10^{38}. Compute the sum of two such biases in single precision, and determine what the softmax returns for a row in which every key is excluded by both masks and for a row in which every key is excluded but some keys by only one mask.

5. code **Attention dropout.** `DotProductAttention` applies dropout with rate p to the attention weights, and the weights it exposes for visualization are those before dropout. Dropout multiplies each kept weight by $1/(1-p)$ and sets the others to zero, so a row of weights no longer sums to 1.
   1. Show that the row sum after dropout has expectation 1 and variance $\frac{p}{1-p}\sum_i \alpha_i^2$, and evaluate the variance for a row that is uniform over l keys and for a one-hot row. What is the output of a one-hot row in the draws in which dropout removes the attended weight?
   2. Use the inputs of Section 10.2.3.2 with dropout 0.5 active, and pass as values a batch of 10×10 identity matrices, so that each output row equals the weights after dropout. Compare the mean and the variance of the row sums over many draws with the prediction computed from the weights before dropout.
6. **Entropy of random scores.** ▯ The scaled entropy curve above stays near 3.7 nats, below the uniform value $\log 64 \approx 4.16$. Fix one query $\mathbf{q} \in \mathbb{R}^d$ and draw m keys with independent standard normal coordinates, as `softmax_weights` does.
   1. Show that the scaled scores $a_i = \mathbf{q}^\top \mathbf{k}_i/\sqrt{d}$ are independent normal variables with mean 0 and variance $s^2 = \|\mathbf{q}\|^2/d$, and compute the expected value of s^2 when $\mathbf{q}$ is drawn as in `softmax_weights`.
   2. The entropy of the weights is $H = \log Z - \sum_i \alpha_i a_i$ with $Z = \sum_j \exp(a_j)$. For large m, replace Z/m and $\sum_i \exp(a_i)\, a_i/m$ by their expected values and derive
   $$H \approx \log m - \frac{s^2}{2}.$$
   Evaluate the approximation for $m = 64$ and $s = 1$.
   3. The approximation needs Z to concentrate around its mean. Compute the relative standard deviation of Z as a function of s and m, and derive from it a condition on s^2 under which the approximation holds. Unscaled scores have $s^2 = \|\mathbf{q}\|^2$, whose expected value is d. Evaluate the relative standard deviation for $m = 64$ at $s^2 = 4$ and $s^2 = 8$, and compare the approximation with the unscaled entropies plotted above for $d = 4$ and $d = 8$.
7. code **Additive scores near zero.** ▯ The additive score Equation 10.13 adds the projected query and the projected key inside a tanh.
   1. Replace tanh by the identity and show that the attention weights then do not depend on the query.
   2. Expand tanh to third order around zero, identify the lowest-order terms through which the weights depend on the query, and state how this dependence scales when $\mathbf{W}_q$ and $\mathbf{W}_k$ are both multiplied by a small factor ϵ.
   3. Multiply `W_q` and `W_k` in the code of Section 10.2.4 by $\epsilon \in \{1, 0.3, 0.1, 0.03\}$, and measure the largest difference between the weight rows of the three queries. Check the scaling you derived, and state what the result for $\epsilon = 1$ implies for additive attention at initialization.
8. [extended] **A learned alignment.** ▯ Bahdanau et al. (2015) observed that attention

weights trained only for translation behave like alignments (Figure 10.5). The same effect appears in a toy model without recurrent networks. Draw source sequences of 8 tokens uniformly from a vocabulary of 16, and let the target at output position j be the source token at position $\pi(j)$ for a fixed permutation π of the positions. Score one learned query per output position against keys that a learned linear map computes from the concatenated one-hot token and one-hot position of each source token, use the one-hot source tokens as values, and train on the negative logarithm of the output entry that belongs to the target token.

1. Implement the model with `DotProductAttention`, train it, and plot the attention weights of one sequence with `d2l.show_heatmaps`. Compare the map with π.
2. Replace the score by the additive rule Equation 10.13 and compare the number of steps that each model needs to reach 99% accuracy.
3. Give queries and keys the width $d \in \{16, 256, 1024\}$ with standard normal initialization, and train with and without the factor $1/\sqrt{d}$. Relate the training curves to the entropy of the weights at initialization.
4. Explain why a fixed query per output position cannot learn a reordering that depends on the tokens, such as swapping two neighbors only when the first belongs to a designated subset of the vocabulary, and state what the query would have to depend on.

10.3 Multi-Head and Cross-Attention

The attention layer of Section 10.2 gives each query one set of weights over the keys and returns one weighted average of the values. That suffices when a query needs one thing from its context. Often it needs several. In the sentence "Alice gave Bob the book", the verb *gave* needs three other words: who gave, who received, and what was given. A single weighted average mixes their three value vectors into one vector of the same width. How can one query read several positions at once without blending what it reads?

We make the question precise with a small task: one query must report the values stored at two positions. Any single set of attention weights, however it is chosen, loses half of the target's variance on this task. *Multi-head attention* (Vaswani et al. 2017) gives each query several sets of weights, each computed in a learned projection of its own, and combines the resulting averages linearly. With the usual choice of head width it has the same number of parameters and the same leading FLOP count as one wide head, and a layer built this way learns to solve the two-value task. The heads do not all need their own keys and values. Sharing them within groups of heads keeps the extra weightings and shrinks the memory that text generation needs. The section ends with self-attention and cross-attention, which are the same operation applied to different inputs.

10.3.1 A One-Mixture Bottleneck with Position-Only Keys

To measure what one weighting per query cannot express, we need a task whose answer we know and whose best single-head solution we can compute. Consider the following task. Two source positions hold value vectors $\mathbf{v}_1, \mathbf{v}_2 \in \mathbb{R}^c$, drawn independently at random for every example. Their keys mark only the *positions*, not the content. A query must report **both** values: the target is the concatenation $\mathbf{t} = [\mathbf{v}_1; \mathbf{v}_2] \in \mathbb{R}^{2c}$.

For this query a single head can only produce softmax weights (α_1, α_2) with $\alpha_1 + \alpha_2 = 1$, the mixture $\mathbf{m} = \alpha_1 \mathbf{v}_1 + \alpha_2 \mathbf{v}_2$, and then a linear readout $\mathbf{W}_o \mathbf{m}$. Since the keys are fixed

position markers, the weights cannot depend on the values. Whatever the head learns, it applies the *same* mixture to every example. How well can it do?

Proposition. Consider one attention head with two fixed, position-only keys, independent standard Gaussian values in $\mathbb{R}^c$, no residual path, and a linear readout. For the copy-both task, the best achievable relative squared error is $1/2$ for every softmax pair (α_1, α_2).

Proof. Given the mixture $\mathbf{m} = \alpha_1 \mathbf{v}_1 + \alpha_2 \mathbf{v}_2$, the best estimate of $\mathbf{v}_1$ is the conditional mean $\mathbb{E}[\mathbf{v}_1 \mid \mathbf{m}] = \frac{\alpha_1}{\alpha_1^2+\alpha_2^2}\,\mathbf{m}$, which is linear, so a linear readout attains it. Its error variance per coordinate is $1 - \frac{\alpha_1^2}{\alpha_1^2+\alpha_2^2} = \frac{\alpha_2^2}{\alpha_1^2+\alpha_2^2}$, and the error for $\mathbf{v}_2$ is $\frac{\alpha_1^2}{\alpha_1^2+\alpha_2^2}$. The two errors sum to 1 per coordinate, while the target has total variance 2 per coordinate. The relative squared error is therefore $1/2$, however the head splits its attention. ■

The weights cannot remove the error because the readout receives a single mixture of the two values, whatever they are. We check the bound by fitting the best linear readout by least squares for several splits, with values of width $c = 8$. The code reports the norm of the readout's error divided by the norm of the target. This relative error is the square root of the relative squared error, so the bound reads $\sqrt{1/2} \approx 0.707$.

```
N, c = 10000, 8
V1, V2 = torch.randn(N, c), torch.randn(N, c)
target = torch.cat([V1, V2], dim=1)

def readout_error(M, target):
    """Best linear readout of target from M, relative error."""
    W = torch.linalg.lstsq(M, target).solution
    return ((M @ W - target).norm() / target.norm()).item()

for a in (0.5, 0.7, 0.9, 1.0):
    M = a * V1 + (1 - a) * V2
    print(f'single head, weights ({a:.1f}, {1 - a:.1f}): '
          f'relative error {readout_error(M, target):.3f}')
print(f'theory: sqrt(1/2) = {0.5 ** 0.5:.3f}')
```

```
single head, weights (0.5, 0.5): relative error 0.707
single head, weights (0.7, 0.3): relative error 0.707
single head, weights (0.9, 0.1): relative error 0.707
single head, weights (1.0, 0.0): relative error 0.707
theory: sqrt(1/2) = 0.707
```

The error stays at about 0.71 across the whole range, as the proposition predicts. Now give the model a second head: two sets of attention weights, whose outputs are concatenated before the readout.

```
for a in (1.0, 0.88):
    M2 = torch.cat([a * V1 + (1 - a) * V2,
                    (1 - a) * V1 + a * V2], dim=1)
    print(f'two heads, weights ({a:.2f}, {1 - a:.2f}) and '
          f'({1 - a:.2f}, {a:.2f}): '
          f'relative error {readout_error(M2, target):.1e}')
```

```
two heads, weights (1.00, 0.00) and (0.00, 1.00): relative error 1.4e-06
```

```
two heads, weights (0.88, 0.12) and (0.12, 0.88): relative error 1.7e-06
```

Two heads bring the error down to the rounding error of the least-squares solver, and the second row does so with *soft* weights $(0.88, 0.12)$. The heads need different weightings, not one-hot ones. They supply two mixtures, the readout inverts the 2×2 mixing matrix, and both values come back exactly. The inversion is well conditioned as long as the two weightings differ appreciably. Exercise 5 follows what happens as they approach each other. Figure 10.6 summarizes the two cases.

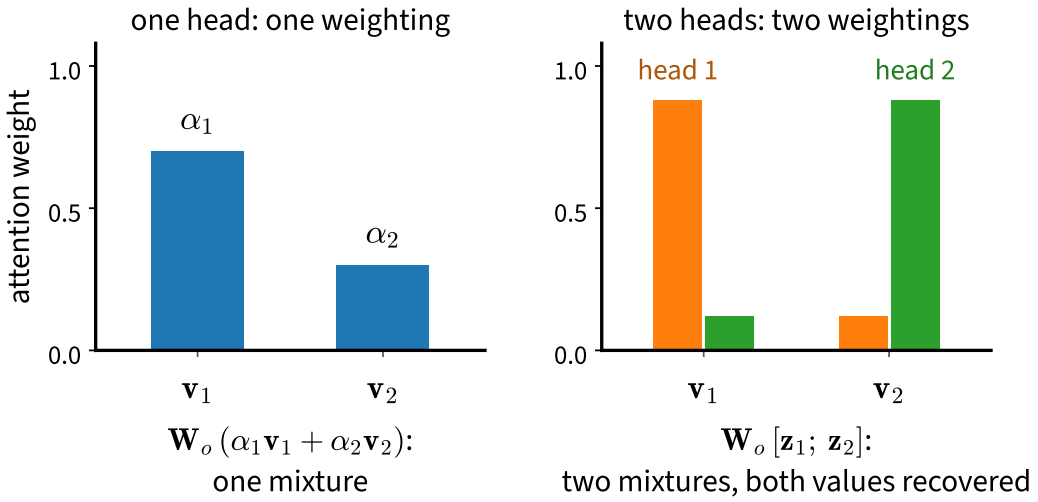


Figure 10.6: One weighting per query against two. A single head serves the copy-both task with one softmax weighting, so its output blends the two requested values. Two heads with different weightings supply two mixtures, from which a linear readout recovers both values.

The bound needs less than this proof used, and it does not depend on the softmax. It holds whenever a linear readout receives one vector of c numbers per query and must produce $2c$ independent ones. No vector of c numbers computed from the values does better, not even a mixture whose weights depend on the values (Exercise 6). Content-dependent keys, more layers, or a residual connection therefore leave the bound in place as long as the readout is linear and receives c numbers. Two heads lift it because they hand the readout $2c$ numbers. A layer that solves the task must do two things: compute several different weightings for one query, and leave room in its output for every value it retrieves.

10.3.2 Multi-Head Attention

10.3.2.1 Projected Heads

Rather than hand-design the heads, we learn them. Given a query $\mathbf{q} \in \mathbb{R}^d$ and key–value pairs $(\mathbf{k}_1, \mathbf{v}_1), \dots, (\mathbf{k}_m, \mathbf{v}_m)$ with $\mathbf{k}_j, \mathbf{v}_j \in \mathbb{R}^d$, head i ($i = 1, \dots, H$) first projects them into its own subspace of width d_h and then runs ordinary scaled dot-product attention f from Section 10.2:

$$\mathbf{z}_i = f\big(\mathbf{W}_i^{(q)}\mathbf{q},\ (\mathbf{W}_i^{(k)}\mathbf{k}_1, \mathbf{W}_i^{(v)}\mathbf{v}_1), \dots, (\mathbf{W}_i^{(k)}\mathbf{k}_m, \mathbf{W}_i^{(v)}\mathbf{v}_m)\big) \in \mathbb{R}^{d_h}, \tag{10.14}$$

with learnable $\mathbf{W}_i^{(q)}, \mathbf{W}_i^{(k)}, \mathbf{W}_i^{(v)} \in \mathbb{R}^{d_h \times d}$. Each head scores with its own projected queries and keys, so it weights the keys by its own notion of relevance, and it moves its own projection of the values. These are the separate mixtures of the construction above, now learned. The layer's output is a learned linear recombination of the concatenated heads:

$$\mathbf{W}_o \begin{bmatrix} \mathbf{z}_1 \\ \vdots \\ \mathbf{z}_H \end{bmatrix} \in \mathbb{R}^d, \qquad \mathbf{W}_o \in \mathbb{R}^{d \times H d_h}. \tag{10.15}$$

The output has width d whatever the number of heads, so a layer of width d returns at most d numbers per query. In the copy-both task it can return both values only if $2c \leq d$. Figure 10.7 shows the layout.

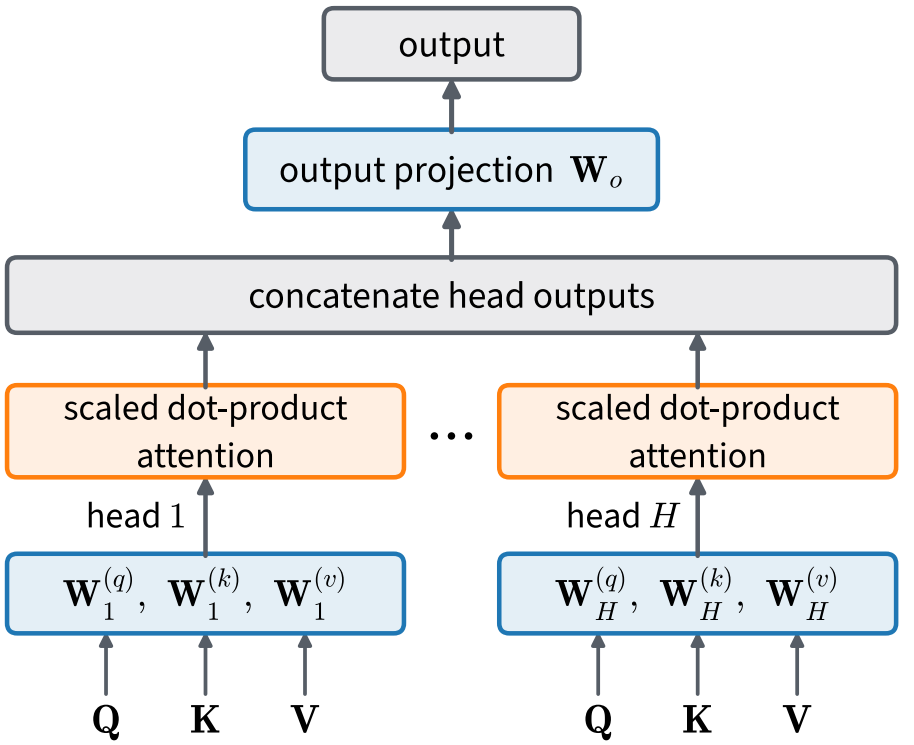


Figure 10.7: Multi-head attention. Every head reads the same queries, keys, and values, projects them into its own subspace, and attends there; the head outputs are concatenated and recombined by one output projection.

10.3.2.2 Parameter and FLOP Counts

The standard choice of head width makes the heads together no more expensive than one wide head. Set $d_h = d/H$, so that the H heads produce $Hd_h = d$ numbers per token, as many as a single head of width d. For self-attention, write the inputs of a sequence of length n as $\mathbf{Q}, \mathbf{K}, \mathbf{V} \in \mathbb{R}^{n \times d}$. Each of the four learned projections is then a $d \times d$ matrix, and each

head receives matrices of shape $n \times d_h$. Counting one multiply–add as two FLOPs, the layer costs

$$\underbrace{8nd^2}_{\text{projections}} + \underbrace{4n^2d}_{\text{scores and mixing}} \quad \text{FLOPs}, \qquad 4d^2 \text{ parameters.} \tag{10.16}$$

Neither count depends on H. The four projection matrices are $d \times d$ however their outputs are sliced into heads, and each head's score and mixing products shrink by the same factor H by which their number grows. The number of heads changes the representation, not the leading operation count. Running time is not quite constant, because more heads mean more and smaller matrix products and softmaxes, which run less efficiently (Exercise 1d). What does change with H is the attention map. Each query now has H distributions over the n keys instead of one.

10.3.2.3 Implementation

The choice $d_h = d/H$ also gives the standard implementation. We compute one query projection of width d and reshape its output into H heads of width d_h, with the heads as an explicit axis: queries, keys, and values become tensors of shape (batch, H, length, d_h). The scaled dot-product attention of Section 10.2 works over any leading axes, so it runs all heads of all sequences in one batched matrix product, with no loop over heads. A mask of shape (batch, queries, keys) receives a head axis of size one and so applies to every head. The argument `num_kv_heads` lets groups of heads share keys and values. We return to it in Section 10.3.2.5; until then it equals the number of heads.

```
class MultiHeadAttention(nn.Module):
    """Multi-head attention with heads as an explicit axis; num_kv_heads <
    num_heads shares each key-value head across a group of query heads."""
    def __init__(self, num_hiddens, num_heads, dropout=0.0, bias=False,
                 num_kv_heads=None):
        super().__init__()
        assert num_hiddens % num_heads == 0, 'heads must divide num_hiddens'
        self.num_heads = num_heads
        self.num_kv_heads = num_kv_heads or num_heads
        assert num_heads % self.num_kv_heads == 0
        kv_hiddens = self.num_kv_heads * (num_hiddens // num_heads)
        self.attention = d2l.DotProductAttention(dropout)
        self.W_q = nn.Linear(num_hiddens, num_hiddens, bias=bias)
        self.W_k = nn.Linear(num_hiddens, kv_hiddens, bias=bias)
        self.W_v = nn.Linear(num_hiddens, kv_hiddens, bias=bias)
        self.W_o = nn.Linear(num_hiddens, num_hiddens, bias=bias)

    def forward(self, queries, keys, values, mask=None):
        # queries: (batch, no. of queries, num_hiddens); keys and values:
        # (batch, no. of key-value pairs, num_hiddens). mask: boolean,
        # broadcastable to (batch, no. of queries, no. of keys), shared by
        # all heads, or to (batch, num_heads, no. of queries, no. of keys)
        q = self.split_heads(self.W_q(queries), self.num_heads)
        k = self.split_heads(self.W_k(keys), self.num_kv_heads)
        v = self.split_heads(self.W_v(values), self.num_kv_heads)
        group = self.num_heads // self.num_kv_heads
```

```
        k, v = k.repeat_interleave(group, 1), v.repeat_interleave(group, 1)
        if mask is not None and mask.dim() == 3:
            mask = mask[:, None]          # The same mask for every head
        # Shape of Y: (batch, num_heads, no. of queries, head width)
        Y = self.attention(q, k, v, mask)
        return self.W_o(self.merge_heads(Y))

    @property
    def attention_weights(self):
        """Weights of the last call: (batch, num_heads, queries, keys)."""
        return self.attention.attention_weights
```

Two reshapes do the work of splitting and merging, as Figure 10.8 shows. `split_heads` cuts the last axis of width Hd_h into H slices of width d_h and moves the head axis in front of the length axis. `merge_heads` reverses both steps, so the output projection sees the head outputs side by side, as in Equation 10.15. Head i therefore owns columns $(i-1)d_h + 1$ to id_h of each projection's output.

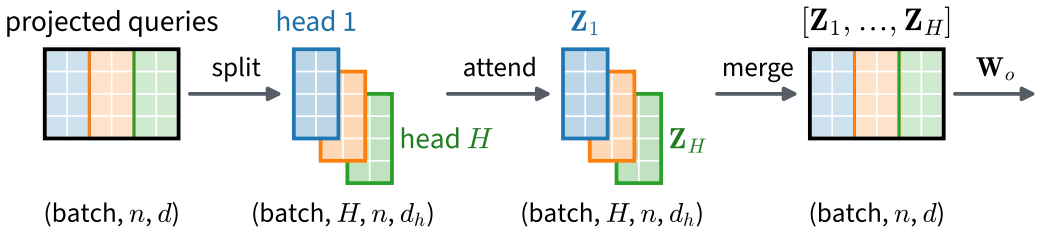


Figure 10.8: Heads as slices of one projection. The projected queries of n tokens are cut into H column slices of width $d_h = d/H$, which become a separate head axis; each head attends on its own, and the head outputs are placed side by side again before the output projection $\mathbf{W}_o$. Keys and values are split in the same way.

```
@d2l.add_to_class(MultiHeadAttention)
def split_heads(self, X, num_heads):
    """(batch, length, num_heads * width) -> (batch, num_heads, length,
    width)."""
    return X.reshape(X.shape[0], X.shape[1], num_heads, -1).transpose(1, 2)

@d2l.add_to_class(MultiHeadAttention)
def merge_heads(self, X):
    """Reverse split_heads: (batch, num_heads, length, width) -> (batch,
    length, num_heads * width)."""
    return X.transpose(1, 2).reshape(X.shape[0], X.shape[2], -1)
```

The parameter count of Equation 10.16 can be checked directly. `d2l.num_params` counts the trainable parameters of a model, and for a layer of width 256 it should give $4 \cdot 256^2 = 262{,}144$ for every number of heads.

```
for num_heads in (1, 2, 4, 8):
    attention = MultiHeadAttention(num_hiddens=256, num_heads=num_heads)
    print(f'H = {num_heads}: {d2l.num_params(attention)} parameters')
```

```
H = 1: 262144 parameters
H = 2: 262144 parameters
H = 4: 262144 parameters
H = 8: 262144 parameters
```

10.3.2.4 Learned Heads on the Copy-Both Task

We now return to the task of Section 10.3.1 with the layer itself instead of hand-built mixtures. A layer of width $d = 16$ receives one query and two key–value pairs. The keys are the standard basis vectors $\mathbf{e}_1$ and $\mathbf{e}_2$, so they mark position only, and the query is $\mathbf{e}_3$ in every example. The values carry the content: $\mathbf{v}_1, \mathbf{v}_2 \in \mathbb{R}^8$, drawn afresh for every example and padded with zeros to width 16. The target $[\mathbf{v}_1; \mathbf{v}_2]$ is the layer's entire output, so the output has room for exactly two values ($2c = d$). The data function draws a minibatch, and the training function fits a layer with Adam on the squared error, drawing a fresh minibatch at every step.

```
def copy_both_data(n, d=16, c=8):
    """One query, two position-only keys, values of width c padded to d."""
    V = torch.randn(n, 2, c)
    keys = torch.eye(d)[:2].expand(n, 2, d)    # e_1, e_2 mark the positions
    query = torch.eye(d)[2].expand(n, 1, d)    # e_3 asks for both values
    values = nn.functional.pad(V, (0, d - c))
    return query, keys, values, V.reshape(n, 1, 2 * c)

def train_copy_both(num_heads, num_steps=2000):
    model = MultiHeadAttention(16, num_heads)
    optimizer = torch.optim.Adam(model.parameters(), lr=1e-2)
    for _ in range(num_steps):
        query, keys, values, target = copy_both_data(256)
        loss = ((model(query, keys, values) - target) ** 2).mean()
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()
    return model
```

We train layers with one, two, and four heads and evaluate each on 10,000 fresh examples, printing the relative error and the weights that each head gives to the two positions.

```
query, keys, values, target = copy_both_data(10000)
for num_heads in (1, 2, 4):
    model = train_copy_both(num_heads)
    with torch.inference_mode():
        err = (model(query, keys, values) - target).norm() / target.norm()
    weights = ' '.join(f'({a:.2f}, {b:.2f})'
                       for a, b in model.attention_weights[0, :, 0].tolist())
    print(f'H = {num_heads}: error {err:.2g}; weights {weights}')
```

```
H = 1: error 0.71; weights (0.51, 0.49)
H = 2: error 1.1e-07; weights (0.91, 0.09) (0.05, 0.95)
H = 4: error 1.3e-07; weights (0.25, 0.75) (0.95, 0.05) (0.93, 0.07) (0.09, 0.91)
```

With one head the relative error stays near $\sqrt{1/2} \approx 0.71$, the bound of the proposition: all 16 outputs of a single head are linear functions of the 8 numbers in one mixture $\alpha_1 \mathbf{v}_1 + \alpha_2 \mathbf{v}_2$. With two or four heads the error falls by more than two orders of magnitude, and the layer returns both values almost exactly. The printed weights show how. The heads learn at least two clearly different weightings of the two positions, and these need not be one-hot. As in the hand-built construction, two different mixtures are all that the output projection needs.

10.3.2.5 Shared Keys and Values

In the copy-both layer the heads had to differ in how they weight the positions, and that difference can come from their query projections alone. Their keys could have been the same, and so could their values, as long as the shared value heads together still carry all c coordinates of each value. With two heads of width $d/2 = c$, a single key–value head suffices. With four heads of width $d/4$, a single shared value head would carry only half of each value, and the bound of Section 10.3.1 returns (Exercise 7).

Sharing matters when a model generates text one token at a time. The model stores the keys and values of all earlier tokens, in every layer and every head, so that it need not recompute them, and it reads all of them again for each new token. Section 11.3 builds this *KV cache* and measures its cost. For GPT-2 small, with 12 layers of 12 heads of width 64, the cache holds $2 \cdot 12 \cdot 12 \cdot 64 = 18{,}432$ numbers per token, or 36 KiB in 16-bit floats, so a context of 1,024 tokens occupies 36 MiB per sequence.

Grouped-query attention (GQA) keeps H query heads but computes only H_{kv} key–value heads, each shared by a group of H/H_{kv} consecutive query heads (Ainslie et al. 2023). Counting from zero, query head i reads key–value head $g(i) = \lfloor iH_{\text{kv}}/H \rfloor$:

$$\mathbf{z}_i = f\big(\mathbf{W}_i^{(q)}\mathbf{q},\ (\mathbf{W}_{g(i)}^{(k)}\mathbf{k}_1, \mathbf{W}_{g(i)}^{(v)}\mathbf{v}_1), \dots, (\mathbf{W}_{g(i)}^{(k)}\mathbf{k}_m, \mathbf{W}_{g(i)}^{(v)}\mathbf{v}_m)\big). \tag{10.17}$$

Each head still computes its own weights, from its own queries, but the heads of a group read the same keys and mix the same values. With $H_{\text{kv}} = H$ this is multi-head attention; with $H_{\text{kv}} = 1$ all heads share one key head and one value head, which is *multi-query attention* (MQA) (Shazeer 2019). The key and value projections shrink from $d \times d$ to $H_{\text{kv}} d_h \times d$, and the cache per token shrinks by the factor H/H_{kv}. The score and mixing FLOPs do not change, because every query head still scores every key.

In `MultiHeadAttention`, `num_kv_heads` sets H_{kv}: `W_k` and `W_v` produce H_{kv} heads, and the forward pass repeats each of them for the query heads of its group. A grouped layer is therefore a multi-head layer whose key and value projections are tied within each group. We check this by copying the weights of a layer with 8 query heads and 2 key–value heads into a full 8-head layer, repeating each key or value projection for the 4 query heads of its group.

```
d, H, H_kv = 256, 8, 2
d_h, group = d // H, H // H_kv
gqa = MultiHeadAttention(num_hiddens=d, num_heads=H, num_kv_heads=H_kv)
mha = MultiHeadAttention(num_hiddens=d, num_heads=H)
with torch.no_grad():
```

```
        mha.W_q.weight.copy_(gqa.W_q.weight)
        mha.W_o.weight.copy_(gqa.W_o.weight)
        for W_gqa, W_mha in ((gqa.W_k, mha.W_k), (gqa.W_v, mha.W_v)):
            W = W_gqa.weight.reshape(H_kv, d_h, d)     # One block per kv head
            W_mha.weight.copy_(W.repeat_interleave(group, 0).reshape(d, d))
    X = torch.randn(2, 10, d)
    print(f'max |GQA - MHA|: {(gqa(X, X, X) - mha(X, X, X)).abs().max():.1e}')
    print(f'parameters: GQA {d2l.num_params(gqa)}, MHA {d2l.num_params(mha)}')
```

```
max |GQA - MHA|: 0.0e+00
parameters: GQA 163840, MHA 262144
```

The two layers agree up to floating-point rounding. The grouped layer has $2 \cdot 256^2 + 2 \cdot 256 \cdot 64 = 163{,}840$ parameters instead of 262,144, and it caches $2 \cdot 2 \cdot 32 = 128$ numbers per token instead of 512.

Sharing keys and values lowers quality somewhat, and by how much depends on how the model was trained. Ainslie et al. converted a trained T5-XXL model to 8 key–value groups and continued pretraining for 5% of the original steps. Averaged over summarization, translation, and question answering, its score fell from 47.2 to 47.1, and inference time per sample fell from 1.51 s to 0.28 s; with a single key–value head the score was 46.6 (Ainslie et al. 2023). DeepSeek-AI trained 7B-parameter models from scratch with each layout, at equal parameter count, and measured larger gaps: MMLU accuracy 45.2 with full heads, 41.2 with 8 groups, and 37.9 with one key–value head (DeepSeek-AI 2024). Most large open models nevertheless use grouped-query attention; gpt-oss, for example, has 64 query heads and 8 key–value heads (OpenAI 2025). The reason is the cache. Its size bounds the minibatch and the context that fit in memory, and reading it dominates the time per generated token. DeepSeek-V2's multi-head latent attention goes further. Per token and layer it caches one low-dimensional vector, from which the keys and values of all heads are reconstructed, plus a small key shared by all heads that carries the token's position (DeepSeek-AI 2024).

10.3.3 Self-Attention and Cross-Attention

`MultiHeadAttention` takes three sequence arguments: queries, keys, and values. Self-attention and cross-attention differ only in where these arguments come from.

10.3.3.1 Self-Attention

In *self-attention* (Z. Lin et al. 2017; Vaswani et al. 2017), one sequence supplies all three arguments. Every token emits a query against every token's key, and each output position is a mixture of values from the same sequence. Given inputs $\mathbf{x}_1, \ldots, \mathbf{x}_n \in \mathbb{R}^d$, the layer returns an equally long sequence $\mathbf{y}_1, \ldots, \mathbf{y}_n$ with

$$\mathbf{y}_i = f\big(\mathbf{x}_i,\ (\mathbf{x}_1, \mathbf{x}_1), \ldots, (\mathbf{x}_n, \mathbf{x}_n)\big) \in \mathbb{R}^d, \tag{10.18}$$

where f is now multi-head attention. Each token's new representation combines information from the whole sequence in a single step. Since the output has the shape of the input, such layers can be stacked, which is how transformers are built.

```
num_hiddens, num_heads = 100, 5
attention = MultiHeadAttention(num_hiddens, num_heads)
batch_size, num_queries = 2, 4
mask = d2l.padding_mask(torch.tensor([3, 2]), num_queries)
X = torch.ones((batch_size, num_queries, num_hiddens))
d2l.check_shape(attention(X, X, X, mask),
                (batch_size, num_queries, num_hiddens))
```

10.3.3.2 Cross-Attention

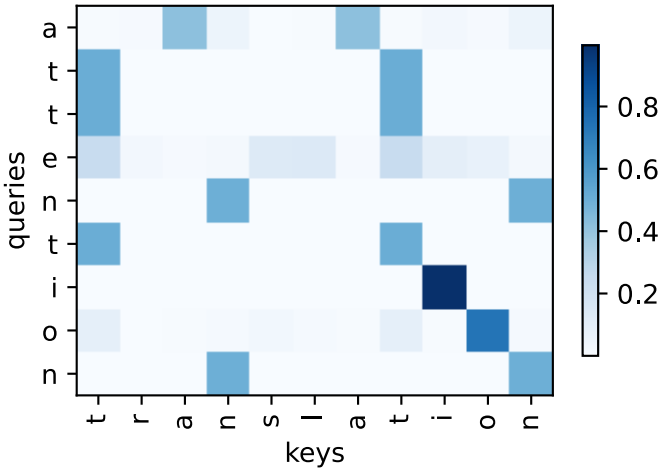
In *cross-attention*, the queries come from a sequence **A**, while the keys and values come from a sequence **B**. The output has the length of **A** and gathers information from **B**. This is the wiring of the original encoder–decoder transformer, where each position of the translation queries the source sentence; of vision–language models such as Flamingo (Alayrac et al. 2022), where text tokens query features of an image; and of the alignment models that introduced learned attention (Section 10.2). All of them call the same function with different arguments.

```
A = torch.ones((batch_size, 4, num_hiddens))
B = torch.ones((batch_size, 9, num_hiddens))
d2l.check_shape(attention(A, B, B), (batch_size, 4, num_hiddens))
```

10.3.3.3 Cross-Attention between Two Words

To see cross-attention at work, give every letter of the alphabet a random embedding of width 32, and let the letters of one word query the letters of another through plain scaled dot-product attention, with no learned projections. Matching letters share an embedding, so their scaled score is close to $\sqrt{32} \approx 5.7$, while unrelated pairs score near zero. The softmax turns this gap into an alignment we can read.

```
words = 'attention', 'translation'
letters = sorted(set(''.join(words)))
emb = torch.randn(len(letters), 32)
queries, keys = (emb[[letters.index(c) for c in w]][None] for w in words)
attention = d2l.DotProductAttention()
attention(queries, keys, keys)
d2l.show_heatmaps(attention.attention_weights[None], xlabel='keys',
                  ylabel='queries', figsize=(4, 3.5), cmap='Blues',
                  xticklabels=words[1], yticklabels=words[0])
```



The map reads row by row. The queries are the letters of "attention", the keys the letters of "translation". Three patterns appear. The `i` and `o` queries find their single partners and attend sharply. The `a`, `t`, and `n` queries find *two* copies each and split their weight between them, because a head cannot tell identical keys apart. This is the averaging of the copy-both task on a small scale. The `e` query, whose letter does not occur in "translation", has no partner. Its weight goes to keys that resemble it only by chance, and none of them receives the near-certain weight of a real match; the softmax must distribute all of its probability mass even when no key matches. Trained models show all three patterns. Language models often handle the third case by sending the leftover weight to the first token of the sequence, which then acts as a sink for attention that has nowhere useful to go (Xiao et al. 2024). A row without a match, or a large weight on such a token, therefore says that no key is relevant, not that the favored key is, which is one reason why attention maps must be read with care as explanations.

None of the layers in this section knows where a token sits. Permuting the keys and values together leaves every output unchanged (Exercise 3). If "translation" were spelled backward, every letter of "attention" would retrieve the same mixture, and only the columns of the map would change places. The next section adds positional information.

10.3.4 Summary

A single attention head gives each query one distribution over the keys and hence one mixture of the values. On the copy-both task, where a query needs two values, its relative squared error is $1/2$ for every choice of weights. Multi-head attention computes H weightings in learned projections of width $d_h = d/H$ and recombines their outputs linearly. Its parameter and leading FLOP counts do not depend on H, and a layer with two heads learns to solve the copy-both task whenever the two values fit into its output. Grouped-query attention lets groups of query heads share keys and values, which shrinks the memory that generation needs by the factor H/H_{kv} at a quality cost that depends on the training setup. Self-attention draws queries, keys, and values from one sequence; cross-attention draws the queries from one sequence and the keys and values from another.

10.3.5 Exercises

1. code **Narrow heads.** With $d_h = d/H$, the leading count of Equation 10.16 does not depend on H, so the number of heads looks free up to $H = d$. Consider one head with scores $\mathbf{q}^\top \mathbf{k}_j / \sqrt{d_h}$ over keys $\mathbf{k}_1, \dots, \mathbf{k}_m \in \mathbb{R}^{d_h}$ and no mask. Call a query sharp on key j when its weights concentrate on $\mathbf{k}_j$ as the query is scaled by a factor that grows without bound.
   1. For $d_h = 1$ and distinct scalar keys, show that every query's weights are monotone in k_j. Conclude that the queries of such a head can be sharp only on the largest or the smallest key.
   2. Show that for $d_h = 2$ the keys and queries can be chosen so that each of n queries is sharp on a key of its own choosing, for every assignment of queries to keys.
   3. Use these two results to state which attention patterns a layer gives up as H approaches d.
   4. Time the forward pass of `MultiHeadAttention` with dropout 0 for self-attention with $d = 256$ and $n = 512$ on a minibatch of 8 sequences for $H \in \{1, 2, 4, 8, 16, 32, 64\}$. Make a warm-up call first and wait for the result before stopping the clock. Report the ratio of the slowest to the fastest configuration. Identify the cost that grows with H and that Equation 10.16 leaves out, and check that it accounts for the trend.
2. code **Heads as slices.** `MultiHeadAttention` computes one projection of width d for each of the queries, keys, and values and splits it into heads with `split_heads`.
   1. Head i owns a slice of the output features of `W_q`, `W_k`, and `W_v`. Determine which input features of `W_o` receive that head's output. Write a forward pass that loops over the heads, applies the sliced projections, and calls `d2l.DotProductAttention` once per head. Verify on random inputs with a padding mask that it matches `MultiHeadAttention` with dropout 0, and extend your loop to `num_kv_heads < num_heads`.
   2. Replace the body of `split_heads` by `X.reshape(X.shape[0], num_heads, X.shape[1], -1)`, which returns a tensor of the same shape. Describe which numbers each "head" now holds, show on random inputs that the output changes, and explain why no shape check can detect the mistake. Which input lengths and widths, if any, make the two versions agree?
3. **Permuting the source.** In cross-attention, the queries come from $\mathbf{A} \in \mathbb{R}^{n \times d}$ and the keys and values from $\mathbf{B} \in \mathbb{R}^{m \times d}$, neither carrying positional information. Let Π be an $m \times m$ permutation matrix.
   1. Show that replacing $\mathbf{B}$ by $\Pi\mathbf{B}$ in both the key and the value argument leaves the output of `MultiHeadAttention` unchanged when no mask is applied, and that replacing it in the key argument alone changes the output in general.
   2. Now let a padding mask mark only the first ℓ rows of $\mathbf{B}$ as valid. Which permutations leave the output unchanged when the mask stays fixed?
   3. Self-attention without positional information is permutation equivariant: permuting the rows of its input permutes the rows of its output in the same way. A translation decoder cross-attends to the output of an encoder built from such self-attention layers. Show that the decoder's outputs are identical for the source sentences "dog bites man" and "man bites dog", and state where word order must enter the model for cross-attention to use it.
4. code **Nonlinear readouts.** ⍰ For Gaussian values, the proof of the proposition in Section 10.3.1 shows that the conditional mean $\mathbb{E}[\mathbf{v}_1 \mid \mathbf{m}]$ is linear in the mixture. Hence no readout, linear or not, improves on the relative error $\sqrt{1/2}$ that `readout_error` mea-

sures.

1. Train a small multilayer perceptron to predict `target` from the mixture $\mathbf{m} = a\,\mathbf{v}_1 + (1-a)\,\mathbf{v}_2$ formed from `v1` and `v2`, for $a = 0.5$ and $a = 0.7$. Compare its relative error on held-out samples with that of the linear readout.
2. Now draw the values uniformly from $\{-1, +1\}^c$. For $a = 0.5$, compute $\mathbb{E}[\mathbf{v}_1 \mid \mathbf{m}]$ coordinate by coordinate, show that it is again linear in $\mathbf{m}$, and conclude that the error stays at $\sqrt{1/2}$ for every readout.
3. For $a = 0.7$ and the same discrete values, show that the map $(v_{1j}, v_{2j}) \mapsto m_j$ is injective. Train the perceptron again and report its held-out error. Explain why this escape from the bound does not contradict the proposition.

5. code **Nearly identical heads.** ⊠ Two heads with weights $(0.5+\epsilon, 0.5-\epsilon)$ and $(0.5-\epsilon, 0.5+\epsilon)$ supply two distinct mixtures, so the linear readout inverts them exactly for every $\epsilon > 0$.
   1. Write the 2×2 mixing matrix, compute its eigenvalues and eigenvectors, and express its condition number as a function of ϵ.
   2. Add independent Gaussian noise of standard deviation $\sigma = 0.01$ to both mixtures and measure the error with `readout_error` for $\epsilon \in \{0.4, 0.1, 0.03, 0.01, 0.003, 0.001\}$. Plot the error against ϵ on logarithmic axes and read off the slope where the curve is straight.
   3. Rewrite the two noisy mixtures as noisy observations of $\mathbf{v}_1 + \mathbf{v}_2$ and of $\mathbf{v}_1 - \mathbf{v}_2$. Use this to explain the slope, to locate the ϵ below which the second head stops helping, and to predict how that point moves with σ. Check the prediction by repeating the sweep at $\sigma = 0.001$, extended down to $\epsilon = 10^{-5}$.
   4. Every head of a freshly initialized layer weights the two positions of the copy-both task close to $(0.5, 0.5)$. Modify `train_copy_both` to record the relative error and the weights of each head every 20 steps, and plot both for $H = 2$. Use the conditioning of the mixing matrix to explain the shape of the error curve in the first few hundred steps. Repeat for $H = 8$ and sort the final heads into those that are nearly one-hot, those that are soft but distinct, and those that nearly duplicate another head.
6. **What the bound counts.** ⊠ The proposition of Section 10.3.1 fixes the softmax weights, but its error bound rests on a weaker fact: the readout receives a single vector $\mathbf{m} \in \mathbb{R}^c$ and must produce the target $\mathbf{t} \in \mathbb{R}^{2c}$.
   1. Let $\mathbf{m} \in \mathbb{R}^c$ be any function of $\mathbf{v}_1$ and $\mathbf{v}_2$, including a mixture whose weights depend on the values, and let $\mathbf{W}_o \in \mathbb{R}^{2c \times c}$ be any matrix. Show that
   $$\mathbb{E}\,\|\mathbf{W}_o\mathbf{m} - \mathbf{t}\|^2 \geq \tfrac{1}{2}\,\mathbb{E}\,\|\mathbf{t}\|^2.$$
   2. A layer of width d with two heads of width $d_h = d/2$ receives the values $\mathbf{v}_1, \mathbf{v}_2 \in \mathbb{R}^c$ padded with zeros to width d. Find the largest c for which it can solve the task exactly. Give the attention weights of each head, value projections $\mathbf{W}_i^{(v)}$, and an output projection $\mathbf{W}_o$ that do so, and compare your weights with those printed in the experiment of Section 10.3.2.4.
   3. Generalize the proposition to k source positions with independent standard Gaussian values $\mathbf{v}_1, \ldots, \mathbf{v}_k \in \mathbb{R}^c$ and the target $[\mathbf{v}_1; \ldots; \mathbf{v}_k]$. Each of H heads applies fixed weights to the values, the weights of head i form row i of a matrix $\mathbf{A} \in \mathbb{R}^{H \times k}$, and a linear readout sees all H mixtures. Show that the best relative squared error is
   $$1 - \frac{\operatorname{rank}(\mathbf{A})}{k}, \tag{10.19}$$

and that softmax weights, whose entries are all positive, can attain $\operatorname{rank}(\mathbf{A}) = \min(H, k)$.

7. code **Sharing keys and values on the copy-both task.** ⊠
   1. Argue that a layer of width d with two query heads and a single key–value head ($H_{\text{kv}} = 1$) can still solve the copy-both task for values of width $c \le d/2$. Which of its parameters must differ between the two heads?
   2. Now let the layer have H query heads of width $d_h = d/H$ and $H_{\text{kv}} < H$ key–value heads. Use the counting of Exercise 6 to show that it can solve the task only if $H_{\text{kv}} d_h \ge c$, and that this condition suffices.
   3. Add an argument `num_kv_heads` to `train_copy_both`. For $d = 16$ and $c = 8$, predict which of the layouts $(H, H_{\text{kv}}) = (2, 1)$, $(4, 2)$, and $(4, 1)$ solve the task, then train them and compare their relative errors with those of the full layers in the experiment of Section 10.3.2.4.
   4. Ainslie et al. convert a trained multi-head model into a grouped one by averaging the key and value projections within each group and then training for a short while (Ainslie et al. 2023). Train a 4-head layer, convert it to $H_{\text{kv}} = 2$ by averaging, and record the relative error right after the conversion and every 10 steps over 200 further training steps. Do the same when you keep the first head of each group instead of the average. Repeat both conversions over at least three runs. Which difference between them holds in every run, and which does not?
   5. For a model with L layers, H query heads of width d_h, and H_{kv} key–value heads, write the cache size in bytes for a context of n tokens stored in 16-bit floats. Evaluate it for GPT-2 small ($L = H = 12$, $d_h = 64$) at $n = 1024$ with $H_{\text{kv}} \in \{12, 4, 1\}$, and for a model with 32 layers, 32 query heads, and 8 key–value heads of width 128 at $n = 32{,}768$.
8. code **Averages under key noise.** ⊠ Let $m = 4$ keys $\mathbf{k}_i \in \mathbb{R}^8$ be drawn as $\mathbf{k}_i \sim \mathcal{N}(\mu_i, \Sigma_i)$ with orthonormal means μ_i, let the values $\mathbf{v}_1, \dots, \mathbf{v}_4$ be the standard basis vectors of $\mathbb{R}^4$, and let the score be $\mathbf{q}^\top \mathbf{k}_i$. When every covariance is $\beta\mathbf{I}$ with vanishingly small β, the keys are nearly orthonormal, so by Exercise 1a of Section 10.1 the query $s\,(\mu_a + \mu_b)$ with a large scale s returns an output $\mathbf{o}$ close to $\frac{1}{2}(\mathbf{v}_a + \mathbf{v}_b)$.
   1. Now let $\Sigma_a = \beta\mathbf{I} + \frac{1}{2}\mu_a\mu_a^\top$, so that $\mathbf{k}_a \approx (1+\gamma)\,\mu_a$ with $\gamma \sim \mathcal{N}(0, \frac{1}{2})$, and leave the other covariances at $\beta\mathbf{I}$. Each draw of the keys then gives $\mathbf{k}_a$ and $\mathbf{k}_b$ different norms, the case of Exercise 1c of Section 10.1: as $\beta \to 0$, the weights on $\mathbf{k}_a$ and $\mathbf{k}_b$ stand in the ratio $e^{s\gamma}$, and the other two weights are negligible at large s. Derive from this the probability, over resamples of the keys, that $\|\mathbf{o} - \frac{1}{2}(\mathbf{v}_a + \mathbf{v}_b)\|$ exceeds 0.1, and evaluate it for $s = 10$ and $s = 100$. What does the probability tend to as $s \to \infty$?
   2. Replace the single query by two queries $\mathbf{q}_1$ and $\mathbf{q}_2$, each with its own softmax and output $\mathbf{o}_1$ or $\mathbf{o}_2$, and return $\frac{1}{2}(\mathbf{o}_1 + \mathbf{o}_2)$. Choose the queries so that the output tends to $\frac{1}{2}(\mathbf{v}_a + \mathbf{v}_b)$ as $s \to \infty$ for most values of γ, and identify the values of γ for which it does not.
   3. Simulate three settings: isotropic covariances with the query $s\,(\mu_a + \mu_b)$, the anisotropic $\mathbf{k}_a$ with that query, and the anisotropic $\mathbf{k}_a$ with the two queries. Use $\beta = 10^{-4}$, $s \in \{10, 100\}$, and 1000 resamples of the keys. Report the median of $\|\mathbf{o} - \frac{1}{2}(\mathbf{v}_a + \mathbf{v}_b)\|$ and the fraction of resamples on which it exceeds 0.1. Compare that fraction for the anisotropic $\mathbf{k}_a$ and the single query with the probability from (a). Explain why raising s from 10 to 100 hurts the single query even when the covariances are isotropic.
   4. Repeat the two single-query settings with the query $-s\sum_{i \notin \{a,b\}} \mu_i$, which lowers

the scores of the other keys instead of raising those of $\mathbf{k}_a$ and $\mathbf{k}_b$. Show that its output does not depend on γ, and determine which failure of the query $s\,(\mu_a + \mu_b)$ remains. Explain why the two queries avoid it.

Adapted from Stanford CS224N, Assignment 4, Problem 1(c) and 1(d).

10.4 Positional Information

The sentences "dog bites man" and "man bites dog" contain the same three tokens. Fed to the self-attention layer of Section 10.3, they produce the same three output vectors, only listed in a different order. Each output averages value vectors with weights computed from content alone, and nothing in that computation records where a token stands. A classifier that pools these outputs cannot tell who bit whom. Attention must therefore be told where each token stands.

There are several ways to do so. Absolute embeddings, designed or learned, add a position vector to each token. Rotary embeddings (RoPE) rotate queries and keys so that every score depends on the offset between two positions. Linear biases (ALiBi) penalize distant keys, and a causal model can even work without any explicit encoding (NoPE). All of them work at the sequence lengths a model was trained on; beyond those lengths they behave very differently. We first prove the symmetry that has to be broken and build each scheme. A small attention-only language model, trained at context 128, then shows what each scheme does at context 512 and how a trained RoPE model can be stretched to the longer context.

10.4.1 Permutation Equivariance

Before choosing a remedy, we state the symmetry precisely, since every scheme in this section is a way of breaking it. Let $\mathbf{X} \in \mathbb{R}^{n \times d}$ hold a sequence of n token representations. At evaluation time, with dropout disabled and with no mask or position-dependent bias, let f be the self-attention layer

$$f(\mathbf{X}) = \operatorname{softmax}\left(\frac{\mathbf{X}\mathbf{W}_q(\mathbf{X}\mathbf{W}_k)^\top}{\sqrt{d}}\right)\mathbf{X}\mathbf{W}_v. \tag{10.20}$$

Proposition (permutation equivariance). For every permutation matrix Π, $f(\Pi\mathbf{X}) = \Pi f(\mathbf{X})$.

Proof. The projections act on each row separately, so $(\Pi\mathbf{X})\mathbf{W} = \Pi(\mathbf{X}\mathbf{W})$ for each of $\mathbf{W}_q, \mathbf{W}_k, \mathbf{W}_v$. The score matrix becomes $\Pi\mathbf{S}\Pi^\top$, and since softmax normalizes each row by itself, it commutes with row and column permutations: $\operatorname{softmax}(\Pi\mathbf{S}\Pi^\top) = \Pi \operatorname{softmax}(\mathbf{S})\,\Pi^\top$. Multiplying by the permuted values, $\Pi \operatorname{softmax}(\mathbf{S})\,\Pi^\top \Pi\mathbf{X}\mathbf{W}_v = \Pi f(\mathbf{X})$ because $\Pi^\top\Pi = \mathbf{I}$. Concatenating heads and applying $\mathbf{W}_o$ row-wise preserves the property, so it holds for multi-head attention too. ■

Permuting the input applies the same permutation to the output: each token's output is unchanged except for its row index. Thus the layer does not use a token's position in the sequence. We check the proposition on the multi-head layer of Section 10.3 by shuffling eight random tokens and comparing the outputs:

```
attention = d2l.MultiHeadAttention(num_hiddens=64, num_heads=4)
X = torch.randn(1, 8, 64)
perm = torch.randperm(8)
Y = attention(X, X, X)
Y_perm = attention(X[:, perm], X[:, perm], X[:, perm])
print(f'max |Y[perm] - Y_perm|: {(Y[:, perm] - Y_perm).abs().max():.2e}')
```

```
max |Y[perm] - Y_perm|: 7.45e-08
```

The two outputs differ only by float32 round-off: the shuffled input produced the shuffled output. Two caveats qualify this result. First, equivariance is a property of the *layer*. Stacking equivariant layers and reading out per token leaves the model equivariant, so depth does not help. Second, the proposition is about unmasked attention. A causal mask singles out each position by how many predecessors it may see, which breaks the symmetry. NoPE, near the end of the section, relies on this asymmetry. The other schemes give each token explicit positional information.

10.4.2 Absolute Position Embeddings

The original transformer attaches a *position vector* $\mathbf{p}_i \in \mathbb{R}^d$ to each position i and feeds $\mathbf{x}_i + \mathbf{p}_i$ into the first layer (Vaswani et al. 2017). Two tokens with identical embeddings at different positions now enter attention as different vectors, and the equivariance proof fails at its first step. The two classic choices of these vectors differ in whether they are designed or learned.

10.4.2.1 Sinusoidal Encodings

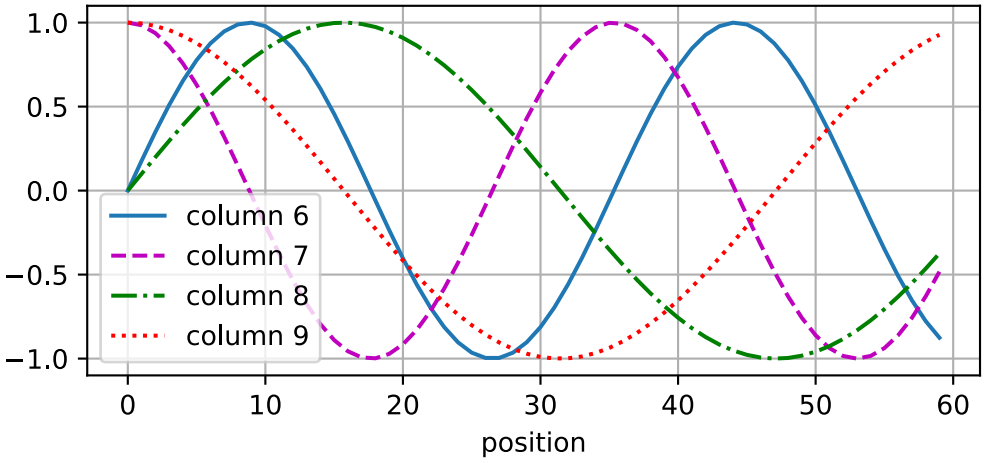
The designed choice fills row i of $\mathbf{P} \in \mathbb{R}^{n \times d}$ with sines and cosines of geometrically spaced frequencies,

$$p_{i,2m} = \sin\left(\frac{i}{10000^{2m/d}}\right), \qquad p_{i,2m+1} = \cos\left(\frac{i}{10000^{2m/d}}\right). \tag{10.21}$$

The design looks arbitrary until we plot it. We implement the table as a small function and look at four adjacent columns.

```
def sinusoidal_encoding(max_len, num_hiddens):
    """The (max_len, num_hiddens) table of sinusoidal position codes."""
    theta = torch.arange(max_len)[:, None] / 10000 ** (
        torch.arange(0, num_hiddens, 2) / num_hiddens)
    return torch.stack([torch.sin(theta), torch.cos(theta)],
                       -1).reshape(max_len, num_hiddens)

P = sinusoidal_encoding(60, 32)
d2l.plot(torch.arange(60), P[:, 6:10].T, xlabel='position',
         figsize=(6, 2.5), legend=[f'column {d}' for d in range(6, 10)])
```



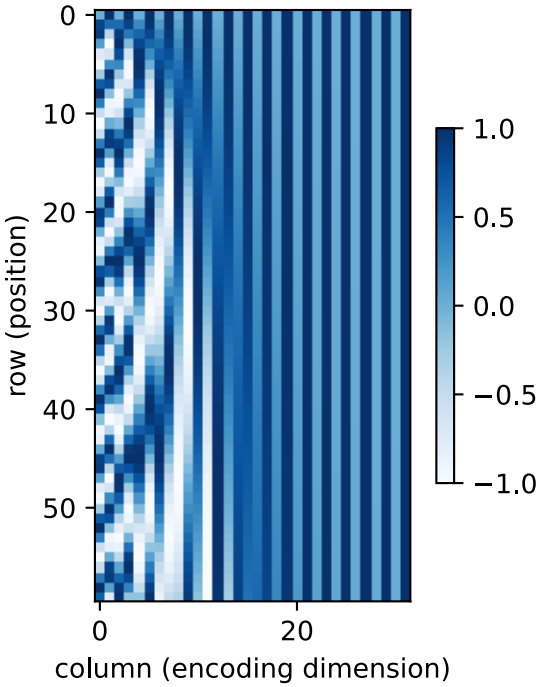
Adjacent column pairs share a frequency (one sine, one cosine), and the frequency falls as the column index grows. This is a continuous version of binary counting. In the binary representations below, the lowest bit flips every number, the next every two numbers, the next every four. Fast dimensions distinguish neighboring positions, whereas slow dimensions distinguish broader regions.

```
for i in range(8):
    print(f'{i} in binary is {i:>03b}')
```

```
0 in binary is 000
1 in binary is 001
2 in binary is 010
3 in binary is 011
4 in binary is 100
5 in binary is 101
6 in binary is 110
7 in binary is 111
```

The heatmap of the whole table shows the same structure in continuous form. Each row is a unique multi-frequency fingerprint of its position. Unlike bits, the values vary smoothly, so nearby positions get nearby encodings.

```
d2l.show_heatmaps(P[None, None], xlabel='column (encoding dimension)',
                  ylabel='row (position)', figsize=(3.5, 4), cmap='Blues')
```



10.4.2.2 Learned Positions

The alternative is to make **P** a trainable embedding table, one free vector per position, as in BERT and GPT-2 (Devlin et al. 2018; Radford et al. 2019). It concedes that we do not know the right encoding and lets gradient descent find one. Its limitation is visible before any training. The table has one row for each position up to some maximum length, and a row that no training example ever used still holds its random initialization. To such a table, position 500 is not "a bit beyond position 128". Its row carries no information at all.

10.4.2.3 Sinusoidal Encodings as Rotations

Designed sinusoids have a relative-position structure absent from a learned table. The original paper chose them because, for every fixed offset δ, the row $\mathbf{p}_{i+\delta}$ is a linear function of the row $\mathbf{p}_i$. The linear map can be written explicitly. Write $\omega_m = 1/10000^{2m/d}$ for the frequency shared by columns $2m$ and $2m+1$. Then

$$\begin{bmatrix} \cos(\delta\omega_m) & \sin(\delta\omega_m) \\ -\sin(\delta\omega_m) & \cos(\delta\omega_m) \end{bmatrix} \begin{bmatrix} p_{i,2m} \\ p_{i,2m+1} \end{bmatrix} = \begin{bmatrix} \sin\left((i+\delta)\omega_m\right) \\ \cos\left((i+\delta)\omega_m\right) \end{bmatrix} = \begin{bmatrix} p_{i+\delta,2m} \\ p_{i+\delta,2m+1} \end{bmatrix}, \tag{10.22}$$

by the angle-addition identities. The map from position i to position $i+\delta$ is a *rotation* of each two-column pair, by an angle $\delta\omega_m$ that depends only on the offset δ, never on i. The encoding thus contains relative position, but only implicitly. The model receives $\mathbf{x}_i + \mathbf{p}_i$ and must learn projections that exploit the rotations. Moreover, a query–key product of two such sums expands into four terms, and only the position–position term retains the rotation directly. RoPE instead builds the relative rotation into the query–key comparison itself.

10.4.3 Rotary Position Embeddings

Rotary position embeddings (RoPE) (Su et al. 2024) skip the addition and act on queries and keys at the point where they meet, the score. We want a transformation $\mathbf{R}_i$, applied to the query at position i and the key at position j, such that the score depends only on the content vectors and the offset $j - i$:

$$(\mathbf{R}_i\mathbf{q})^\top(\mathbf{R}_j\mathbf{k}) = \mathbf{q}^\top\mathbf{R}_i^\top\mathbf{R}_j\mathbf{k} \overset{!}{=} \mathbf{q}^\top\mathbf{R}_{j-i}\mathbf{k}. \tag{10.23}$$

The requirement $\mathbf{R}_i^\top\mathbf{R}_j = \mathbf{R}_{j-i}$ pins down the kind of map we need. With $i = j$ it reads $\mathbf{R}_i^\top\mathbf{R}_i = \mathbf{R}_0 = \mathbf{I}$ (position 0 is left as it is), so every $\mathbf{R}_i$ is orthogonal. With $j = 0$ it gives $\mathbf{R}_{-i} = \mathbf{R}_i^\top$, and substituting this back yields $\mathbf{R}_i\mathbf{R}_j = \mathbf{R}_{i+j}$. The $\mathbf{R}_i$ form a one-parameter group of orthogonal maps, and planar rotations are the simplest such group. RoPE therefore reuses the sinusoidal frequencies ω_m, now *multiplicatively*. It splits the d dimensions of each query into $d/2$ pairs, where d is now the width of one attention head, and rotates pair m of the query at position i by the angle $i\,\omega_m$ (and likewise for keys),

$$\mathbf{R}_i = \begin{bmatrix} R(i\omega_0) & & \\ & \ddots & \\ & & R(i\omega_{d/2-1}) \end{bmatrix}, \qquad R(\alpha) = \begin{bmatrix} \cos\alpha & -\sin\alpha \\ \sin\alpha & \cos\alpha \end{bmatrix}. \tag{10.24}$$

Figure 10.9 shows one pair plane. Shifting both positions by the same amount rotates query and key together, preserving their angle and their dot product. The score therefore depends on position only through the offset. Rotations also preserve norms, so RoPE never inflates or shrinks a token's content. The geometric spacing of the frequencies sets how fast position turns each pair. The fast pairs separate neighboring positions sharply, while the slow pairs barely move across a sentence and carry content largely unchanged. Only queries and keys are rotated; values pass through untouched. RoPE is the positional scheme of most open-weights language models released since 2023.

`rope_tables` computes the cosines and sines of the angles, and `rope` rotates the even/odd feature pairs with them. The argument `offset` shifts every position by the same amount, which we use to test the invariance. Two optional arguments change the angles themselves: `base` replaces the constant 10000 in the frequencies, and `scale` multiplies the angle of each pair by its own factor. Both keep their defaults until Section 10.4.4.4 stretches a trained model to longer contexts. A caller that rotates many tensors at the same positions can compute the tables once and pass them to `rope` as `tables`; cached generation does this in Section 11.3.1.1.

```
def rope_tables(n, d, offset=0, base=10000.0, scale=None,
                device=None):
    """Cosines and sines, each (n, d/2), of the angles of the positions
    offset, ..., offset + n - 1."""
    pos = torch.arange(n, dtype=torch.float32, device=device) + offset
    omega = base ** (-torch.arange(0, d, 2, device=device) / d)
    if scale is not None:  # One angle factor per pair, shape (d/2,)
        omega = omega * scale
    theta = pos[:, None] * omega[None, :]
    return torch.cos(theta), torch.sin(theta)

def rope(x, offset=0, base=10000.0, scale=None, tables=None):
```

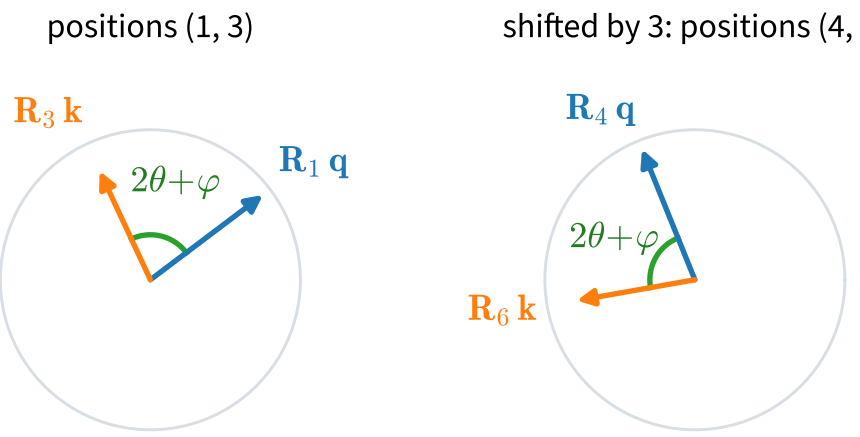


Figure 10.9: Rotary embeddings rotate each two-dimensional feature pair of the query and the key by an angle proportional to position. Shifting both positions by the same amount, here by 3, rotates both vectors together and leaves the angle between them ($2\theta + \varphi$ in both panels), and hence the attention score, unchanged. Here $\theta = \omega_m$ is the angle by which pair m turns per position, and φ is the angle between the unrotated query and key.

```
    """Rotate feature pairs of x (..., num_steps, d) by position angles;
    tables from rope_tables take the place of offset, base, and scale."""
    if tables is None:
        tables = rope_tables(*x.shape[-2:], offset, base, scale, x.device)
    # Angles in fp32, tables in the dtype of x: bf16 queries stay bf16
    cos, sin = (u.to(x.dtype) for u in tables)
    x1, x2 = x[..., 0::2], x[..., 1::2]
    return torch.stack([x1 * cos - x2 * sin,
                        x1 * sin + x2 * cos], -1).flatten(-2)
```

To check Equation 10.23 numerically, we shift all positions jointly; the full RoPE score matrix should not change. For contrast, we run the same test on the additive sinusoidal encoding, where the four-term expansion leaves absolute position in the score.

```
T, d = 32, 64
q, k = torch.randn(T, d), torch.randn(T, d)

scores = lambda off: rope(q, off) @ rope(k, off).T / math.sqrt(d)
for off in (1, 17, 480):
    print(f'RoPE, all positions shifted by {off:>3}: max score change '
          f'{(scores(off) - scores(0)).abs().max():.1e}')

P = sinusoidal_encoding(600, d)
added = lambda off: (q + P[off:off+T]) @ (k + P[off:off+T]).T / math.sqrt(d)
print(f'additive sinusoidal, shifted by  17: max score change '
      f'{(added(17) - added(0)).abs().max():.1e}')
```

```
RoPE, all positions shifted by   1: max score change 1.9e-06
```

```
RoPE, all positions shifted by  17: max score change 1.4e-06
RoPE, all positions shifted by 480: max score change 1.9e-05
additive sinusoidal, shifted by  17: max score change 2.6e+00
```

At every shift, RoPE's scores change only by float32 round-off, several orders of magnitude below the scores themselves. This holds even for a shift far beyond where any additive table would end. The additive encoding, in contrast, moves the scores by an amount comparable to the scores themselves. With RoPE, relative position is built into the architecture rather than learned from data. This property does not guarantee accurate predictions at offsets absent from the training data; the next experiment tests such predictions.

10.4.4 Extrapolation Beyond the Training Length

Every scheme above breaks the permutation symmetry on sequences as long as those seen in training. The schemes differ on longer ones. A length-extrapolation test trains at context L and compares perplexity at L with perplexity at, for example, $4L$. Two more schemes were designed for this setting.

10.4.4.1 ALiBi and NoPE

ALiBi (attention with linear biases) (Press et al. 2022) encodes no positions at all. It instead subtracts a distance penalty from every attention score. For query position i and key position $j \le i$ in head h of H,

$$\text{score}_{ij} = \frac{\mathbf{q}_i^\top \mathbf{k}_j}{\sqrt{d}} - m_h\,(i-j), \qquad m_h = 2^{-8h/H}. \tag{10.25}$$

Each head discounts distant tokens at its own geometric rate. Head 1 uses a steep penalty and head H a much smaller one. For $H = 4$ the slopes are $1/4, 1/16, 1/64, 1/256$. The formula $2^{-8h/H}$ is the paper's choice for head counts that are powers of two, and the paper gives a separate rule for other H. The bias depends only on distance, so it is defined for every context length. A distance of 400 is a valid input to the bias, not an unallocated table index. The drawback is a recency prior built into every score, which large content scores can offset but training cannot remove. With slope $1/4$, a key 40 positions back already loses a factor e^{-10} of its weight, so the steep heads see only a short window whatever the context length.

NoPE (no positional encoding) uses the asymmetry that the proposition leaves out (Kazemnejad et al. 2023). A causal model can do without explicit position representations, because the mask already breaks the symmetry. Position 0 attends over one token, position 100 over a hundred and one. The number of tokens competing in each softmax is itself positional information, and a decoder can in principle recover relative order from it. Because it uses neither a table nor a rotation, NoPE has no explicit position range. Its effectiveness depends on how much usable positional signal the mask supplies.

10.4.4.2 An Attention-Only Language Model

To compare extrapolation, we train a deliberately minimal model that the chapter analyzes again later: a token embedding, a stack of residual causal attention blocks, and an output head tied to the embedding (Press and Wolf 2017). There is no feed-forward network and no normalization layer, and the attention projections carry no bias terms by default (a `bias`

switch restores them). Only attention mixes information across positions, which makes the model easy to analyze in Section 10.6. It is the attention-only cousin of the `TinyLM` of Section 9.5.2.

The positional scheme is the one constructor argument that the experiment varies. `pos` takes `'learned'`, `'sinusoidal'`, `'rope'`, `'alibi'`, or `'none'`, and the constructor allocates what the scheme needs. With RoPE, the model keeps the base and the per-pair angle factors of `rope` as attributes, `rope_base` and `rope_scale`, so that they can be changed after training. The method `stream` runs the residual stream through the blocks and also returns the attention weights of each block, which Section 10.6 inspects.

```
class TinyCharLM(nn.Module):
    """Attention-only language model."""
    def __init__(self, vocab_size, num_hiddens=128, num_heads=4, num_blks=2,
                 pos='rope', max_len=512, bias=False):
        super().__init__()
        assert pos in ('learned', 'sinusoidal', 'rope', 'alibi', 'none')
        self.num_heads, self.pos = num_heads, pos
        self.token_emb = nn.Embedding(vocab_size, num_hiddens)
        nn.init.normal_(self.token_emb.weight, std=0.02)
        if pos == 'learned':
            self.pos_emb = nn.Embedding(max_len, num_hiddens)
            nn.init.normal_(self.pos_emb.weight, std=0.02)
        if pos == 'sinusoidal':
            self.register_buffer('P', sinusoidal_encoding(max_len,
                                                          num_hiddens))
        if pos == 'rope':  # Base and per-pair angle factors of the rotation
            self.rope_base = 10000.0
            self.register_buffer('rope_scale',
                                 torch.ones(num_hiddens // num_heads // 2))
        self.blks = nn.ModuleList([nn.ModuleDict(dict(
            qkv=nn.Linear(num_hiddens, 3 * num_hiddens, bias=bias),
            proj=nn.Linear(num_hiddens, num_hiddens, bias=bias)))
            for _ in range(num_blks)])

    def stream(self, X, num_blks=None):
        """Residual stream after the first num_blks blocks (default: all),
        and the attention weights of those blocks."""
        H, maps = self.embed(X), []
        for blk in self.blks[:num_blks]:
            out, weights = self.attend(blk, H)
            H, maps = H + out, maps + [weights]
        return H, maps

    def forward(self, X):
        return F.linear(self.stream(X)[0], self.token_emb.weight)  # Tied

    def attention_weights(self, X):
        """Per-block attention maps, each (batch, num_heads, T, T)."""
        return self.stream(X)[1]
```

The absolute schemes act once, on the input: `embed` adds the learned or the sinusoidal table to the token embeddings. With the sinusoidal table, the token embeddings are multiplied

by $\sqrt{d}$ before the table is added, as in the original transformer. Without this factor the table, whose entries have magnitude about 0.7, would outweigh embeddings that start with entries of about 0.02 by a factor of 35. With it, the factor is about 3.

```
@d2l.add_to_class(TinyCharLM)
def embed(self, X):
    """Token embeddings plus the absolute position code, if any."""
    H = self.token_emb(X)
    if self.pos == 'learned':
        H = H + self.pos_emb(torch.arange(X.shape[1], device=X.device))
    if self.pos == 'sinusoidal':  # Scale tokens up to the table
        H = H * math.sqrt(H.shape[-1]) + self.P[:X.shape[1]]
    return H
```

The relative schemes act inside every block, on the queries, keys, and scores. `attend` is causal multi-head attention in which RoPE rotates the queries and the keys, and ALiBi subtracts the distance penalty of Equation 10.25 from the scores. With every other scheme it is plain causal attention.

```
@d2l.add_to_class(TinyCharLM)
def alibi(self, T, device):
    """ALiBi bias -m_h (i - j) of every head, (num_heads, T, T)."""
    h = torch.arange(1, self.num_heads + 1, device=device)
    slopes = 2.0 ** (-8.0 * h / self.num_heads)
    pos = torch.arange(T, device=device, dtype=torch.float32)
    return slopes[:, None, None] * (pos[None, :] - pos[:, None])

@d2l.add_to_class(TinyCharLM)
def attend(self, blk, H):
    """Output of attention block blk on the stream H, and its weights."""
    B, T, D = H.shape
    q, k, v = blk['qkv'](H).chunk(3, -1)
    q, k, v = (u.reshape(B, T, self.num_heads, -1).transpose(1, 2)
               for u in (q, k, v))
    if self.pos == 'rope':
        q = rope(q, base=self.rope_base, scale=self.rope_scale)
        k = rope(k, base=self.rope_base, scale=self.rope_scale)
    scores = q @ k.transpose(-2, -1) / math.sqrt(q.shape[-1])
    if self.pos == 'alibi':
        scores = scores + self.alibi(T, H.device)
    weights = d2l.masked_softmax(scores, d2l.causal_mask(T, device=H.device))
    out = (weights @ v).transpose(1, 2).reshape(B, T, D)
    return blk['proj'](out), weights
```

10.4.4.3 The Experiment

We train one model per scheme on `d2l.WikiText103`, the bytes of Wikipedia articles that Section 9.5.2 introduced. Each model trains at context length 128 for 3000 steps of `d2l.train_steps`, the fixed-step training loop of Section 9.5. The tokens are bytes, so each model has an embedding table of 256 rows. The learned and sinusoidal tables are allocated out to `max_len` $= 512$ positions so that the models can be evaluated on longer sequences, but only the first 128 rows of the learned table receive gradients.

```
data = d2l.WikiText103(batch_size=64, num_steps=128)
schemes = ('learned', 'sinusoidal', 'rope', 'alibi', 'none')
models = {}
for pos in schemes:
    model = TinyCharLM(len(data.vocab), pos=pos)
    losses = d2l.train_steps(model, data,
                             torch.optim.AdamW(model.parameters(), lr=1e-3,
                                               weight_decay=0.0), 3000)
    models[pos] = model
    print(f'{pos:>10}: final training loss {d2l.final_loss(losses):.2f}')
```

```
   learned: final training loss 2.32
sinusoidal: final training loss 1.93
      rope: final training loss 2.01
     alibi: final training loss 2.33
      none: final training loss 2.45
```

Every model was trained on length-128 sequences. We evaluate validation perplexity at that training context, $n = 128$, and at two and four times this length ($n = 256$ and 512). The first point is in range; the other two measure extrapolation. All three read the same 1.1 million bytes of the validation split. We cut it into consecutive windows of 128 bytes with `data.windows` and regroup them into windows of 256 and 512, so that only the context available to each prediction changes. `d2l.lm_perplexity` (Section 9.5) sums the cross-entropy over all predicted tokens and converts it to a perplexity once at the end.

The table lists the perplexity of each model at the three context lengths.

```
train_len, batch = data.num_steps, 16
contexts = (train_len, 2 * train_len, 4 * train_len)
block = batch * max(contexts)  # Whole minibatches at every context
num_bytes = (len(data.val) - 1) // block * block
X, Y = data.windows(data.val, range(0, num_bytes, train_len))
eval_sets = {}
for n in contexts:  # The same bytes, regrouped into windows of n
    Xn, Yn = X.reshape(-1, n), Y.reshape(-1, n)
    eval_sets[n] = [(Xn[i:i + batch], Yn[i:i + batch])
                    for i in range(0, len(Xn), batch)]
ppls = {pos: [d2l.lm_perplexity(models[pos], eval_sets[n]) for n in contexts]
        for pos in schemes}
print(f'{"context":>10}: ' + ''.join(f'{n:>8}' for n in contexts))
for pos in schemes:
    print(f'{pos:>10}: ' + ''.join(f'{p:8.1f}' for p in ppls[pos]))
```

```
   context:      128      256      512
   learned:     10.1     20.0     28.7
sinusoidal:      6.8     17.6     30.6
      rope:      7.4     17.4     43.9
     alibi:     10.2     10.1     10.1
      none:     11.6     11.7     11.8
```

At the training length, every explicit scheme gives a lower perplexity than NoPE, and the sinusoidal table and RoPE give the lowest. Beyond the training length the schemes split into two groups. The learned table, the sinusoidal table, and RoPE all degrade at context 512, by

up to several-fold, each for its own reason. The learned table reaches rows beyond position 127 that still hold their random initialization. The sinusoidal table supplies rows whose slow columns take values never seen in training. RoPE adds no position vector, but its slow pairs have periods longer than 128 positions. At offsets beyond 127 they turn through angles they never reached during training, and the projections were never fit to scores computed from those angles.

The other two schemes hold their level. ALiBi stays nearly flat because its bias depends only on distance. NoPE, which has no table and no angles to run out of, stays flat from a higher starting point. A flat curve, however, does not mean that a model *uses* the longer context. ALiBi's steep heads hardly look past the training window at all, as Exercise 4 measures.

10.4.4.4 Extending a Trained Model's Context

Only some of RoPE's pairs meet anything new beyond the training length. A pair whose period $2\pi/\omega_m$ is at most 128 positions completes a full turn within the training offsets, so every angle it produces at offset 400 lies within $\omega_m/2$ of an angle it already took during training. For the head width 32 of `TinyCharLM`, six of the 16 pairs are fast in this sense, with periods from 6 to 112 positions. The other ten pairs, whose periods exceed the training length, reach angles at long offsets that they never took during training. Because the angles vary continuously, they can be *rescaled* so that the deployed range maps into the trained one. Multiplying every angle by $128/512$ makes the offset 4δ look like the familiar δ, so the model interpolates instead of extrapolating. This is *position interpolation* (S. Chen et al. 2023), which extended Llama models from 2,048 to 32,768 tokens with at most 1000 fine-tuning steps.

Uniform rescaling also compresses the fast pairs, which had already seen every angle, and blurs the angles they use to tell neighbors apart. YaRN (Peng et al. 2024) therefore interpolates only the slow pairs, leaves the fast ones as trained, ramps between the two groups, and adds a softmax temperature. NTK-aware scaling, which the YaRN paper also describes, reaches a similar split by changing only the base $b = 10^4$. To grow the context by a factor s (here $s = 4$), it replaces b by $b' = b\, s^{d/(d-2)}$, which leaves the fastest pair unchanged and divides the frequency of the slowest pair by exactly s. Training with a larger base from the outset is the trained form of the same idea. Llama 2 Long, for example, raised the base from 10^4 to 5×10^5 in continued pretraining on long sequences (Xiong et al. 2024).

The attributes `rope_base` and `rope_scale` of `TinyCharLM` let us try two of these rescalings on the trained RoPE model without any training: a uniform factor $128/512$ for all pairs, and the same factor for the slow pairs only.

```
rope_model = models['rope']
d_head = 2 * rope_model.rope_scale.numel()
periods = (2 * math.pi
           * rope_model.rope_base ** (torch.arange(0, d_head, 2) / d_head))
slow = periods > train_len
print(f'{int(slow.sum())} of {d_head // 2} pairs have periods above '
      f'{train_len}')
ones = torch.ones(d_head // 2)
shrink = train_len / max(contexts)  # Longest context into the trained one
factors = {'unscaled': ones, 'all pairs': ones * shrink,
           'slow pairs': torch.where(slow, ones * shrink, ones)}
```

```
    for name, factor in factors.items():
        rope_model.rope_scale.copy_(factor)
        ppl = [d2l.lm_perplexity(rope_model, eval_sets[n]) for n in contexts]
        print(f'{name:>10}: ' + ''.join(f'{p:8.1f}' for p in ppl))
    rope_model.rope_scale.copy_(ones);
```

```
10 of 16 pairs have periods above 128
  unscaled:      7.4    17.4    43.9
 all pairs:     48.4    48.7    48.7
slow pairs:      8.0     8.4     8.7
```

Rescaling only the slow pairs brings the perplexity at context 512 far below that of the unscaled model, and close to its level at the training length. The uniform factor fails on this model. It compresses the fast pairs too, blurs the distinctions between neighboring bytes that they were trained to make. The perplexity jumps already at the training length and stays at that high level at every context, so at 512 it gains nothing reliable over leaving the angles alone and remains far above the rescaling of the slow pairs. This is the defect that YaRN's split into fast and slow pairs removes. Even so, at context 512 neither rescaled model reaches the perplexity that the unscaled model has at the training length, because no weight was trained on compressed angles. The published recipes therefore pair the rescaling with a short fine-tune at the new length, which Exercise 8 carries out on this model.

10.4.5 Summary

Unmasked attention is permutation equivariant, so a model built from it cannot tell where a token stands unless it is told. Absolute embeddings add a position vector to each token. The vector comes from a designed sinusoidal table, whose shifts are rotations of column pairs, or from a learned table, whose rows are trained only at the positions seen in training. RoPE rotates query and key pairs so that every score depends on the offset alone, ALiBi subtracts a per-head linear distance penalty, and NoPE relies on the asymmetry of the causal mask. Trained at context 128, all explicit schemes beat NoPE in range, but the learned table, the sinusoidal table, and RoPE degrade at four times that length, while ALiBi and NoPE stay flat. For RoPE only the slow pairs reach unseen angles at long offsets, and rescaling the angles (position interpolation, YaRN, a larger base) maps a longer context back into the trained range.

10.4.6 Exercises

1. **Distinct positions, distinct codes.** Let $\mathbf{p}_i$ be row i of the sinusoidal table Equation 10.21 of width d, with the frequencies $\omega_m = 1/10000^{2m/d}$ of Equation 10.22.
    1. Show that
    $$\mathbf{p}_i^\top \mathbf{p}_{i+\delta} = \sum_{m=0}^{d/2-1} \cos(\delta\, \omega_m),$$
    derive $\|\mathbf{p}_i - \mathbf{p}_{i+\delta}\|^2$ from it, and conclude that both quantities depend on the offset δ alone.
    2. Conclude that $\mathbf{p}_i = \mathbf{p}_{i'}$ only if $(i - i')\,\omega_m$ is a multiple of 2π for every m, and that no two distinct integer positions share a row, whatever the value of `max_len`.

3. The fastest pair has period 2π, so it nearly repeats at integer offsets close to multiples of 2π. Find the offset $\delta < 512$ at which its angle comes closest to a multiple of 2π, and give the residual angle.
4. For $d = 32$, evaluate $\|\mathbf{p}_i - \mathbf{p}_{i+\delta}\|$ at this offset and at $\delta = 1$, and find the offset between 1 and 511 at which the distance is smallest. Which pairs keep the two positions of the near repeat apart? Relate the answer to the binary counting of Section 10.4.2.1.

2. code **Untrained rows.** The table `pos_emb` of `models['learned']` has `max_len = 512` rows, of which training at context 128 used only the first 128.

1. Construct a new `TinyCharLM` with `pos='learned'`, copy its table `pos_emb` before training, and train it as in the experiment of Section 10.4.4.3. Explain why rows 128 to 511 of the trained table are bitwise equal to the copy, and check the claim. By what factor would decoupled weight decay with coefficient 0.1 at the learning rate 10^{-3} have shrunk these rows over the 3000 training steps?
2. Compute the cosine similarities between all 512 rows of the trained table and display them with `d2l.show_heatmaps`. Describe the three blocks of the result: trained rows against trained rows, untrained against untrained, and trained against untrained.
3. By Exercise 1, the dot products of the sinusoidal rows depend only on $i-j$, so the sinusoidal similarity matrix is constant along each diagonal $i - j = \text{const}$. For the trained rows 0 to 127, compute the standard deviation of the cosine similarities along each diagonal and average it over the diagonals. Compare the average with the standard deviation of the diagonal means across diagonals, and decide whether training produced a code whose similarities are approximately invariant to shifts.

3. **Additive codes in the score.** With the additive encoding of Section 10.4.2.1, a first-layer head of Equation 10.20 scores the inputs at positions i and j as

$$\text{score}_{ij} = (\mathbf{x}_i + \mathbf{p}_i)^\top \mathbf{A}\,(\mathbf{x}_j + \mathbf{p}_j), \qquad \mathbf{A} = \mathbf{W}_q \mathbf{W}_k^\top / \sqrt{d}. \tag{10.26}$$

1. Expand the score into its four terms, and state which of them depend on i, which on j, and which on both.
2. Let $\mathbf{J}$ be the rotation of the plane by $\pi/2$. Using Equation 10.22, show that the position–position term $\mathbf{p}_i^\top \mathbf{A}\,\mathbf{p}_j$ depends only on $j - i$ when $\mathbf{A}$ is block diagonal with 2×2 blocks $a_m \mathbf{I} + b_m \mathbf{J}$, one per frequency pair. Exercise 1 treats $\mathbf{A} = \mathbf{I}$. Then show that a single nonzero entry of $\mathbf{A}$ that couples two different frequency pairs destroys this property.
3. Determine when the content–position term $\mathbf{x}_i^\top \mathbf{A}\,\mathbf{p}_j$ is unchanged for every shift of j. Use the answer to explain the score change of the additive encoding in the shift test of Section 10.4.3, and why RoPE has no content–position term.

4. code **Where extrapolation fails.** The perplexities of Section 10.4.4.3 at context 512 average the loss over all positions of each sequence.

1. Show that in each of the five models the logits at positions 0 to 127 of a length-512 sequence agree, up to floating-point round-off, with those computed from its first 128 tokens alone. What does this imply about the positions at which the perplexity changes between context 128 and context 512?
2. Compute the validation loss at every position of the length-512 sequences for the five models, average it over windows of 16 positions, and plot it against position.

For each scheme, where does the curve leave the level it holds below position 128, and does the loss jump at position 128 or grow gradually?

3. Restrict every query of `models['rope']` to the 128 most recent keys, itself included, by adding a band constraint to the causal mask in a subclass that overrides `attend`, copy the trained weights into it, and repeat the measurement. Compare the windowed curve beyond position 128 with the unwindowed curve below it. Do offsets beyond 127 account for the degradation of RoPE at context 512?
4. ALiBi stays approximately flat at context 512. For the query at position 511 and equal content scores for all keys, compute the fraction of the weight that each of the slopes of Equation 10.25 with $H = 4$ leaves on the distances $i - j \geq 128$, none of which occurred in training. Measure the same fraction for the four heads of each block of `models['alibi']` at context 512, and decide whether the stability of ALiBi at that length comes from never attending beyond the training range.

5. **Biases that keep the symmetry.** ◻ Suppose that a fixed matrix $\mathbf{B} \in \mathbb{R}^{n \times n}$ is added to the score matrix $\mathbf{S}$ of the proof before the softmax, so that the layer Equation 10.20 computes $\text{softmax}(\mathbf{S} + \mathbf{B})\,\mathbf{X}\mathbf{W}_v$.

   1. Show that the proposition still holds for every permutation matrix Π with $\Pi^\top \mathbf{B} \Pi = \mathbf{B}$.
   2. Find all matrices $\mathbf{B}$ that satisfy this condition for every permutation, and describe how each of them changes the attention weights.
   3. Without the causal mask, the ALiBi bias of Equation 10.25 extends symmetrically to $B_{ij} = -m_h\,|i-j|$. Determine the permutations that preserve this matrix, and state what the result implies for an unmasked layer that receives this bias and no other positional signal. Explain why the causal ALiBi of Equation 10.25 does not share this symmetry.

6. **Offset preference in RoPE.** ◻ Split a query $\mathbf{q}$ and a key $\mathbf{k}$ into the $d/2$ two-dimensional pairs $\mathbf{q}^{(m)}, \mathbf{k}^{(m)} \in \mathbb{R}^2$ on which Equation 10.24 acts, and let $\delta = j - i$ be the offset of the key position j from the query position i.

   1. Show that the RoPE score Equation 10.23 equals

   $$\sum_{m=0}^{d/2-1} \|\mathbf{q}^{(m)}\|\,\|\mathbf{k}^{(m)}\| \cos(\delta\,\omega_m - \varphi_m), \tag{10.27}$$

   where φ_m is the angle from $\mathbf{k}^{(m)}$ to $\mathbf{q}^{(m)}$.
   2. Choose the angles φ_m so that every pair contributes its maximum at the offset $\delta = -1$, the previous token. With all products $\|\mathbf{q}^{(m)}\|\,\|\mathbf{k}^{(m)}\|$ equal to c, show that the score, as a function of δ, is c times a shifted copy of the dot product $\mathbf{p}_i^\top \mathbf{p}_{i+\delta}$ of Exercise 1 for a sinusoidal table of width d.
   3. Take $d = 32$ and the query at position 127, which attends to the offsets $\delta = -127, \dots, 0$. Let the attention logits equal these scores, with the factor $1/\sqrt{d}$ absorbed into c. Find the c at which the offset -1 receives half of the softmax weight, and identify the offsets that compete most with it. Which pairs set the gap between the score at -1 and the scores at its neighbors?

7. code **The RoPE base.** ◻ In Equation 10.24 the frequencies are $\omega_m = 10000^{-2m/d}$, and the constant 10000 is a choice.

   1. For the head width 32 of `TinyCharLM`, compute the periods $2\pi/\omega_m$ of the 16 pairs

for the bases 100, 10^4, and 10^6, and count the pairs that complete at least one full rotation within 128 positions. Which period does not depend on the base?
2. Retrain the `'rope'` variant of `TinyCharLM` with bases 100 and 10^6 by setting its attribute `rope_base` after construction. Train two models for each base and one more for base 10^4, so that differences between bases can be compared with the variation across runs. Compare perplexity at 128, 256, and 512.
3. Explain from the counts why the base is expected to matter more at length 512 than at 128, and check the expectation against your results.

8. [extended] **Position interpolation.** ⊠ The last cell of Section 10.4.4.4 rescales the angles of `models['rope']` without any training. This problem explains its result, adds the base change of NTK-aware scaling, and completes the recipe with a fine-tune.

1. Show that a pair that completes a full rotation within the training offsets 0 to 127 has already met, to within $\omega_m/2$ modulo 2π, every angle that a longer offset produces. Compute the gap of Exercise 6 between the score at offset -1 and the scores at its neighbors for $d = 32$, before the rescaling, after the uniform rescaling by $128/512$, and after the rescaling of only the pairs whose period exceeds 128 positions. Relate the result to the perplexities at context 128 in the cell.
2. Show that the base $b' = b\, s^{d/(d-2)}$ leaves ω_0 unchanged and divides $\omega_{d/2-1}$ by s. Set `rope_base` of `models['rope']` to this base for $s = 4$, evaluate at 128, 256, and 512 without training, and compare with the two rescalings of the cell.
3. Fine-tune for a few hundred steps at context 512, on `d2l.WikiText103(batch_size=16, num_steps=512)`, with each of the two rescalings of the cell, each starting from a copy of the trained `models['rope']`. Re-evaluate at 128, 256, and 512, and compare with the unscaled model and with the ALiBi model of the experiment in Section 10.4.4.3 at the same lengths.

10.5 The Cost of Attention

The language model of Section 10.4 was trained on contexts of 128 bytes. Language models in use today read whole reports, code repositories and books, often tens or hundreds of thousands of tokens at once. Self-attention lets every token read every other token in a single layer, and the cost is one score for every pair of positions. How does this cost grow with the length of the context, and how can it be reduced?

Doubling the context quadruples the number of pairs. A context of 8,192 tokens already has 67 million query–key pairs per head, and storing their scores in single precision takes 256 MiB for one head of one sequence. Arithmetic and memory both grow with the square of the sequence length, and at long contexts this quadratic term dominates the cost of the whole layer.

We first compare self-attention with the convolutional and recurrent layers of earlier chapters, count its cost, and check the count against measurements on a GPU. The rest of the section develops three ways to reduce that cost. The first changes only the order of the computation. It computes exact attention one block of keys at a time, without ever storing the full matrix, and FlashAttention fits this schedule to the memory hierarchy of a GPU. The second scores fewer pairs: each token attends only to a window of recent tokens, and depth restores the long-range paths. The third replaces the softmax by a similarity kernel that factorizes, which turns attention into a recurrent network whose state is a matrix. We end by timing the three side by side.

```
from torch.nn.attention import SDPBackend, sdpa_kernel
from torch.nn.attention.flex_attention import create_block_mask, flex_attention
```

10.5.1 Quadratic Time and Memory

10.5.1.1 CNNs, RNNs, and Self-Attention

Consider the problem of mapping a sequence of n tokens, each a d-dimensional vector, to another sequence of the same shape. Three familiar architectures can perform this mapping: a one-dimensional CNN (Chapter 6), an RNN (Section 8.4), and self-attention (Section 10.3). Figure 10.10 draws all three as graphs over the sequence. Two properties of these graphs matter beyond raw arithmetic. The number of *sequential* operations bounds how much of the work can run in parallel on modern hardware. The *maximum path length* counts how many layer transformations connect two positions. Path length is a property of the architecture. A short path avoids multiplying many recurrent Jacobians, but it does not by itself guarantee that training finds a long-range dependency.

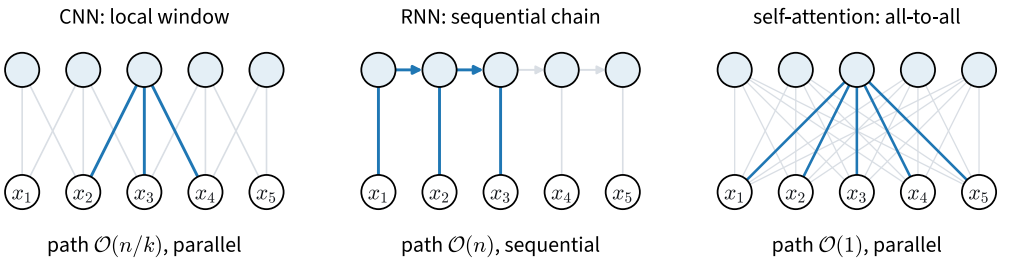


Figure 10.10: Sequence-layer connectivity for five tokens. Highlighted edges show how the inputs reach the third output in one layer: a convolution with kernel size $k = 3$ sees its neighbors, a recurrence passes information step by step along the chain, and self-attention connects every position directly. Below each panel are the path length between distant tokens and whether the outputs are computed in parallel.

A convolutional layer with kernel size k and d channels in and out costs $O(knd^2)$ operations: each of the n positions mixes k neighbors through a $d \times d$ map. All positions are computed at once, so there are $O(1)$ sequential steps, but the receptive field grows by only $k - 1$ positions per layer. Connecting a pair of tokens n apart takes a stack of depth $O(n/k)$. An RNN requires $O(nd^2)$ for the sequence, one $d \times d$ state update per token, but the updates are ordered. It takes $O(n)$ sequential steps, and a signal from the first token reaches the last only through $O(n)$ applications of the cell.

Self-attention first projects the inputs at cost $O(nd^2)$, then computes an $n \times d$ by $d \times n$ product and an $n \times n$ by $n \times d$ product at cost $O(n^2 d)$. These matrix multiplications have $O(1)$ sequential depth, and any token can attend to any other in one layer, giving path length $O(1)$. Table 10.1 collects the accounting.

Table 10.1: Per-layer cost of mapping n tokens of dimension d to n tokens of dimension d.

layer type	complexity	sequential operations	maximum path length
convolution (kernel k)	$O(knd^2)$	$O(1)$	$O(n/k)$
recurrence	$O(nd^2)$	$O(n)$	$O(n)$
self-attention	$O(nd^2 + n^2d)$	$O(1)$	$O(1)$

Self-attention has constant sequential depth and path length, but its mixing term is quadratic in n. The table includes the query, key, value, and output projections so that all three rows count the complete layer. When n and d are comparable, the linear projection and quadratic mixing terms have similar order. For $n \gg d$, the mixing term dominates. The remainder of this section studies ways to reduce its memory use or arithmetic cost.

10.5.1.2 Counting FLOPs

The leading constants make this comparison more precise. One attention layer receives n queries, keys, and values of dimension d and computes $\text{softmax}(\mathbf{Q}\mathbf{K}^\top/\sqrt{d})\mathbf{V}$ (Equation 10.12). The score matrix $\mathbf{Q}\mathbf{K}^\top$ costs $2n^2d$ floating-point operations, because each of its n^2 entries is a length-d dot product and we count one multiply–add as two FLOPs. The softmax costs a few operations per entry, $O(n^2)$, and the value mixing $\text{softmax}(\cdot)\mathbf{V}$ another $2n^2d$. In total

$$\underbrace{4n^2d}_{\text{scores and mixing}} + O(n^2) \quad \text{FLOPs.} \tag{10.28}$$

The four projections around attention in a multi-head layer cost $8nd^2$, which is linear in n, and the head count drops out of both terms (Equation 10.16). Setting $4n^2d = 8nd^2$ locates the crossover: the quadratic part dominates the layer as soon as $n > 2d$. GPT-2 small, with $d = 768$ and contexts of 1,024 tokens, sits below the crossover, so its projections still cost more than its scores. A model with $d = 4096$ at $n = 131{,}072$ sits far beyond it. There $n/2d = 16$, so an attention layer does sixteen times as much score-and-mix work as projection work.

10.5.1.3 Counting Memory

Arithmetic is only one part of the cost, and at long contexts memory is often the binding constraint. A naive implementation materializes the score matrix and its softmax. These are two $n \times n$ buffers of activations, $8n^2$ bytes in single precision, per head and per sequence in the batch. During training the attention weights are also saved for the backward pass. The projections, by contrast, keep only $O(nd)$ activations. At $n = 8192$ and a head dimension of $d_h = 64$, the setting of the experiments below, the inputs $\mathbf{Q}, \mathbf{K}, \mathbf{V}$ occupy 6 MiB together, while the two score buffers occupy 512 MiB. At a context of $n = 131{,}072$ tokens a *single* attention map in fp32 would take $n^2 \times 4$ bytes ≈ 69 GB (per head, per sequence), which is why no system at that scale ever stores one.

We check the formula against a measurement on one head. The function `attention` computes $\text{softmax}(\mathbf{Q}\mathbf{K}^\top/\sqrt{d})\mathbf{V}$ through the full score matrix, and `d2l.peak_memory` of Section 5.7

reports the temporary memory that a call needs. The prediction is the two $n \times n$ float buffers, $8n^2$ bytes.

PyTorch's caching allocator reports what a computation allocated. `reset_peak_memory_stats` clears the high-water mark and `max_memory_allocated` reads it back.

```
def attention(Q, K, V):
    """softmax(Q K^T / sqrt(d)) V through the full score matrix."""
    scores = Q @ K.transpose(-2, -1) / math.sqrt(Q.shape[-1])
    return torch.softmax(scores, dim=-1) @ V
```

We measure a single head of width 64 as n doubles.

```
d_h = 64
for n in [2048, 4096, 8192, 16384]:
    Q = torch.randn(n, d_h, device=d2l.try_gpu())
    mem = d2l.peak_memory(attention, Q, Q, Q)
    print(f'n = {n:5d}: measured {mem/2**20:7.1f}'
          f' MiB, predicted 8n^2 B = {8*n*n/2**20:7.1f} MiB')
```

```
n =  2048: measured    40.6 MiB, predicted 8n^2 B =    32.0 MiB
n =  4096: measured   129.0 MiB, predicted 8n^2 B =   128.0 MiB
n =  8192: measured   514.0 MiB, predicted 8n^2 B =   512.0 MiB
n = 16384: measured  2052.0 MiB, predicted 8n^2 B =  2048.0 MiB
```

The measurement matches $8n^2$ bytes up to a small extra workspace, and doubling n quadruples it. At $n = 16{,}384$, a single fp32 head already needs 2 GiB of temporary storage.

The running time has the same quadratic dependence. We time the forward pass as n doubles, after warm-up calls. On an accelerator, kernel launches return before the work is done, so we synchronize before reading the clock. `d2l.Benchmark` of Section 5.7 does both and reports the average time per call.

```
for n in [2048, 4096, 8192, 16384]:
    Q = torch.randn(n, d_h, device=d2l.try_gpu())
    print(d2l.Benchmark(lambda: attention(Q, Q, Q), desc=f'n = {n:5d}'))
```

```
n =  2048: 0.10 ms/call
n =  4096: 0.46 ms/call
n =  8192: 1.85 ms/call
n = 16384: 7.39 ms/call
```

At small n, the time does not yet follow the operation count: the GPU is not saturated, launch overhead weighs in, and the smallest score matrices still fit in the GPU's on-chip cache. Beyond a few thousand tokens, each doubling of n multiplies the time by about four, the growth of the $4n^2d$ operation count.

10.5.2 Exact Attention Without the Matrix

10.5.2.1 Blockwise Online Softmax

The full $n \times n$ matrix need not be stored. The apparent obstacle to computing attention piecewise is the softmax. Each weight $\exp(a_j)/\sum_{j'} \exp(a_{j'})$ depends on *all* scores of its row through the normalizer and, for numerical safety, through the row maximum subtracted before exponentiation. Both quantities can be maintained *online*, the way one computes a running mean (Milakov and Gimelshein 2018). Process the keys in blocks and keep, per query, three running statistics: the maximum m seen so far, the sum s of exponentials rescaled to that maximum, and the output accumulator $\mathbf{o}$. A new block of scores $\{a_j\}$ with values $\{\mathbf{v}_j\}$ updates them as

$$\begin{aligned} m' &= \max\left(m, \max_j a_j\right), \\ s' &= s\, e^{m-m'} + \sum_j e^{a_j - m'}, \\ \mathbf{o}' &= \mathbf{o}\, e^{m-m'} + \sum_j e^{a_j - m'}\, \mathbf{v}_j, \end{aligned} \tag{10.29}$$

and after the last block, $\mathbf{o}/s$ is exactly $\mathrm{softmax}(\mathbf{a})\mathbf{V}$. Whenever a new block raises the maximum, the factor $e^{m-m'}$ rescales all quantities accumulated under the old maximum, so no approximation is made. Figure 10.11 shows the resulting schedule. One $n \times c$ stripe of the score matrix is in memory at a time, next to a running state of size $O(nd)$ (Rabe and Staats 2021).

10.5.2.2 A Chunked Implementation

From here on we work in the causal setting of a language model. Excluded scores are overwritten with the most negative finite value of the dtype, and `d2l.causal_mask` supplies the mask, as in Section 10.2. We first implement a naive reference, which accepts any number of leading axes such as heads, and then a chunked version for one head. The chunked implementation applies Equation 10.29 to all queries at once and processes one key block per iteration.

```
def causal_attention(Q, K, V):
    """Reference: causal attention with the full score matrix."""
    scores = Q @ K.transpose(-2, -1) / math.sqrt(Q.shape[-1])
    mask = d2l.causal_mask(Q.shape[-2], device=Q.device)
    scores.masked_fill_(~mask, torch.finfo(Q.dtype).min)
    return torch.softmax(scores, dim=-1) @ V

def chunked_attention(Q, K, V, chunk_size=512):
    """Exact causal attention, one n-by-chunk_size block at a time."""
    n, d = Q.shape
    m = torch.full((n, 1), torch.finfo(Q.dtype).min, device=Q.device)
    s = torch.zeros(n, 1, device=Q.device)
    O = torch.zeros(n, V.shape[-1], device=Q.device)
    pos_q = torch.arange(n, device=Q.device)[:, None]
    for start in range(0, n, chunk_size):
        Kc, Vc = K[start:start + chunk_size], V[start:start + chunk_size]
```

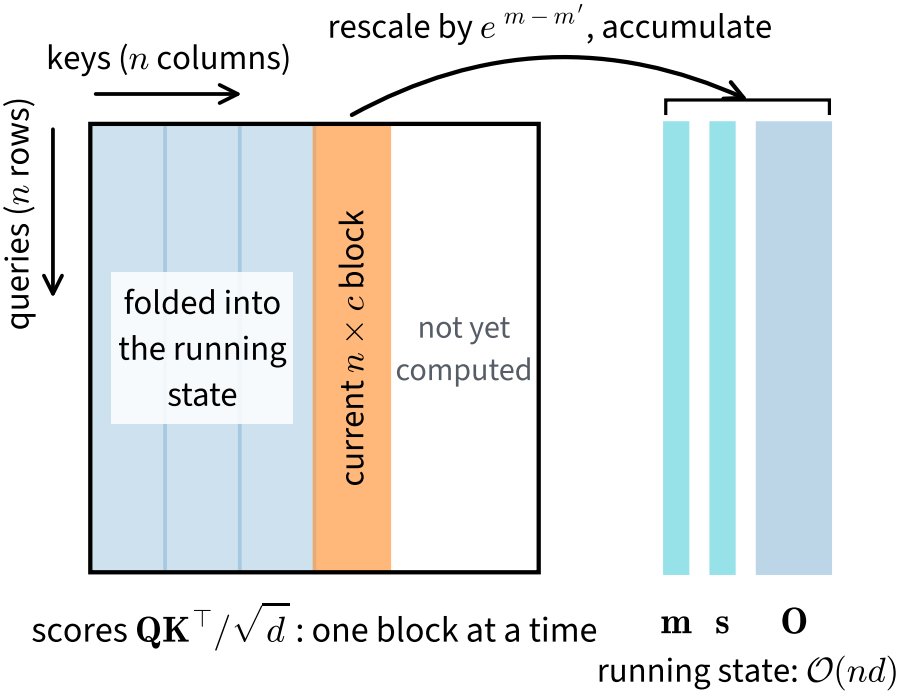


Figure 10.11: Online softmax processes the score matrix one key block at a time. Of the scores, only the current $n \times c$ stripe is in memory, next to the running statistics $\mathbf{m}$, $\mathbf{s}$, $\mathbf{O}$; the full $n \times n$ matrix is never materialized.

```
        pos_k = torch.arange(start, start + Kc.shape[0], device=Q.device)
        scores = Q @ Kc.T / math.sqrt(d)
        scores.masked_fill_(pos_k[None, :] > pos_q, torch.finfo(Q.dtype).min)
        m_new = torch.maximum(m, scores.max(dim=-1, keepdim=True).values)
        scale = torch.exp(m - m_new)                  # Rescale the past
        P = torch.exp(scores - m_new)
        s = s * scale + P.sum(dim=-1, keepdim=True)
        O = O * scale + P @ Vc
        m = m_new
    return O / s
```

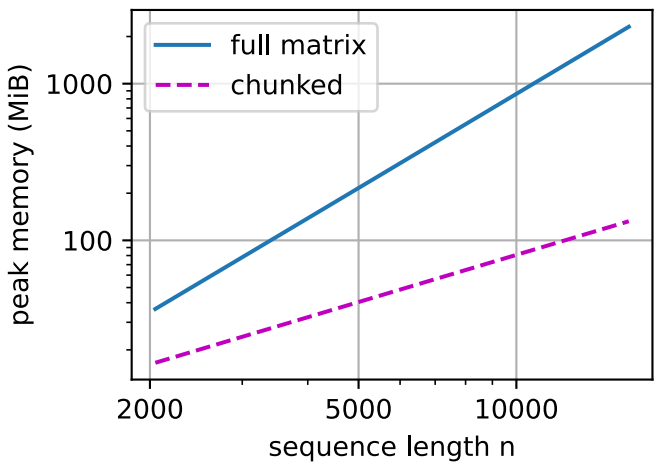
The chunked version is meant to be exact, so we check it against the reference.

```
n, d_h = 2048, 64
Q, K, V = (torch.randn(n, d_h, device=d2l.try_gpu()) for _ in range(3))
err = (chunked_attention(Q, K, V) - causal_attention(Q, K, V)).abs().max()
print(f'maximum deviation: {float(err):.2e}')
```

```
maximum deviation: 2.38e-07
```

The two computations agree up to floating-point rounding, at the level of 10^{-7} to 10^{-6} in fp32. Chunking changes the order of evaluation, not the definition of attention. The chunked version holds $n \times c$ scores at a time instead of $n \times n$, so its footprint should grow linearly in n rather than quadratically.

```
lengths, mems = [2048, 4096, 8192, 16384], [[], []]
for n in lengths:
    Q = torch.randn(n, d_h, device=d2l.try_gpu())
    mems[0].append(d2l.peak_memory(causal_attention, Q, Q, Q) / 2**20)
    mems[1].append(d2l.peak_memory(chunked_attention, Q, Q, Q) / 2**20)
d2l.plot(lengths, mems, 'sequence length n', 'peak memory (MiB)',
         legend=['full matrix', 'chunked'], xscale='log', yscale='log')
```



At $n = 16{,}384$ the full-matrix implementation needs about 2 GiB of scratch memory. The

chunked one needs more than an order of magnitude less, and the gap doubles with every further doubling of n. The result and the $4n^2d$ FLOP count are unchanged. Only the summation order and the memory schedule differ.

10.5.2.3 Memory Traffic as the Computational Bottleneck

Chunking also reduces data movement. On a modern GPU, memory bandwidth often limits attention more than arithmetic throughput. An accelerator can execute hundreds of arithmetic operations in the time it takes to fetch one float from off-chip memory. Naive attention writes the n^2 score matrix to slow memory and reads it back for the softmax and again for the value mixing. The chunked schedule keeps each stripe in fast on-chip memory, finishes all work on it, and never writes it out. FlashAttention (Dao et al. 2022) is this algorithm fitted to the hardware, with tile sizes matched to on-chip SRAM and softmax statistics kept in registers. During training, the backward pass recomputes the stripes instead of storing the weights, because the extra arithmetic takes less time than the memory traffic it avoids. The result is exact attention that is faster *and* needs asymptotically less memory.

Later versions keep the algorithm and change its schedule. FlashAttention-2 divides by the normalizer only once at the end, as `chunked_attention` does, and cuts each sequence into blocks of queries that run in parallel, so that a single long sequence still keeps the whole GPU busy (Dao 2024). FlashAttention-3 overlaps data movement, matrix products, and exponentials on newer GPUs and adds an 8-bit floating-point mode (Shah et al. 2024). In 16-bit precision every version computes exact attention; only the schedule follows the hardware. Frameworks provide it as a fused kernel: `torch.nn.functional.scaled_dot_product_attention` in PyTorch and `jax.nn.dot_product_attention` in JAX. We compare it with the naive reference on a minibatch of 2 sequences with 8 heads of dimension 64 in half precision at $n = 8192$.

```
def fused_attention(Q, K, V):
    """Causal attention in PyTorch's fused FlashAttention kernel."""
    with sdpa_kernel(SDPBackend.FLASH_ATTENTION):
        return F.scaled_dot_product_attention(Q, K, V, is_causal=True)

B, H, n = 2, 8, 8192
X = torch.randn(B, H, n, d_h, device=d2l.try_gpu(), dtype=torch.float16)
for name, f in [('naive', causal_attention), ('fused', fused_attention)]:
    t = d2l.Benchmark(lambda: f(X, X, X)).time
    mem = d2l.peak_memory(f, X, X, X)
    print(f'{name}: {t*1e3:6.2f} ms, peak memory {mem/2**20:7.1f} MiB')
```

```
naive:  26.05 ms, peak memory  4176.0 MiB
fused:   1.22 ms, peak memory    16.5 MiB
```

The fused kernel is several times faster than the naive implementation, and it needs almost no temporary memory where the naive one needs gigabytes. Both gains come from keeping the score blocks out of off-chip memory. In practice, dense attention on sequences beyond a few hundred tokens therefore runs in a fused kernel.

`sdpa_kernel(SDPBackend.FLASH_ATTENTION)` restricts `scaled_dot_product_attention` to the FlashAttention backend, which accepts half and bfloat16 inputs of shape (batch, heads, length, head width). Without the context manager, PyTorch picks a backend by itself and falls back to an unfused computation when no fused kernel supports the inputs.

10.5.3 Windowed and Sparse Attention

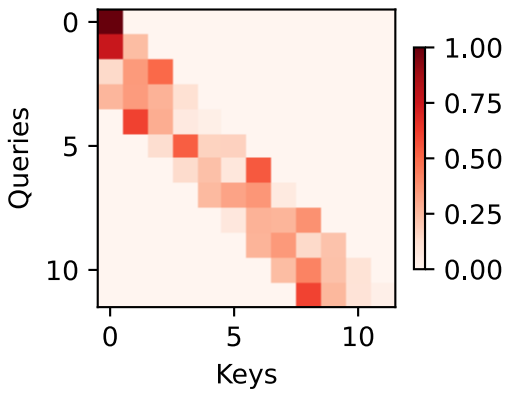
10.5.3.1 Sliding-Window Attention

FlashAttention removes the n^2 *memory*, but every query still scores every key, so the arithmetic remains quadratic. To reduce it too, we must decide that some query–key pairs are not worth scoring. The simplest useful rule is locality: each query attends only to the w most recent positions. With the masks of Section 10.2, the causal mask becomes a band: causal, and at most $w - 1$ positions back.

```
def window_mask(n, w, device=None):
    """Causal band: query i may attend to keys i - w + 1, ..., i."""
    i = torch.arange(n, device=device)
    return d2l.causal_mask(n, device=device) & (i[:, None] - i[None, :] < w)
```

With a window of 4, each row of random scores keeps its weight on the query's own position and the three before it:

```
scores = torch.randn(12, 12)
d2l.show_heatmaps(d2l.masked_softmax(scores, window_mask(12, 4))[None, None],
                  xlabel='Keys', ylabel='Queries')
```



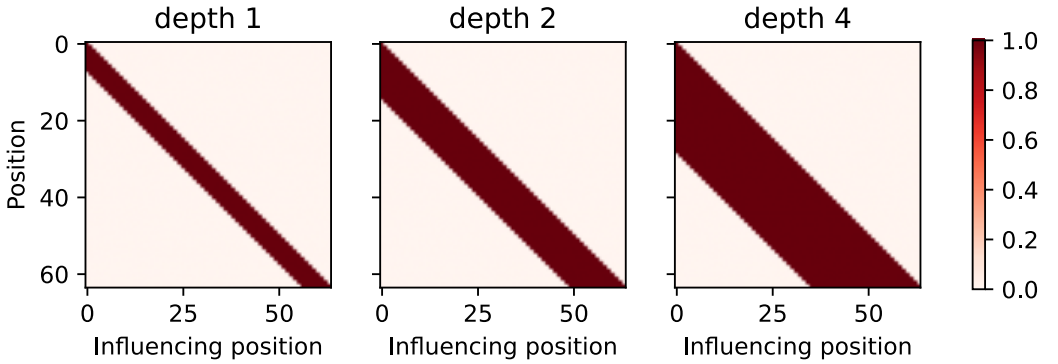
10.5.3.2 Receptive Field Growth with Depth

A window of width w excludes direct interactions beyond w tokens, but depth increases the receptive field as in CNNs (Chapter 6). A query in layer 2 attends to keys that are themselves layer-1 outputs, each of which already summarizes its own window. Information therefore hops up to $w - 1$ positions per layer, and after L layers a token's receptive field spans $1 + L(w - 1)$ positions. We can compute this directly. The band mask, read as an adjacency matrix, composes across layers by boolean matrix multiplication, and its L-th power says which positions can influence which through L layers.

```
n, w, depths = 64, 8, [1, 2, 4]
band = window_mask(n, w).float()
reach, maps = torch.eye(n), []
```

```
for L in range(1, max(depths) + 1):
    reach = ((reach @ band) > 0).float()
    if L in depths:
        maps.append(reach)
        print(f'depth {L}: last query reaches {int(reach[-1].sum())} '
              f'positions (formula: {min(n, 1 + L * (w - 1))})')
d2l.show_heatmaps(torch.stack(maps)[None], xlabel='Influencing position',
                  ylabel='Position', titles=[f'depth {L}' for L in depths],
                  figsize=(8, 3.2))
```

```
depth 1: last query reaches 8 positions (formula: 8)
depth 2: last query reaches 15 positions (formula: 15)
depth 4: last query reaches 29 positions (formula: 29)
```



The count matches the formula, and the heatmaps show the band widening layer by layer. Mistral 7B uses this growth: it attends through a window of 4096 in each of its 32 layers, for a reach beyond a hundred thousand tokens (Jiang et al. 2023). Table 10.1 describes what is given up. A stack of windows has path length $O(n/w)$ instead of $O(1)$, so with w in the thousands it assumes that few dependencies need more than a small number of hops.

More recent models therefore interleave windowed layers with a few layers of full attention, called *global* layers. Gemma 3 follows every five windowed layers of width 1024 with one global layer (Gemma Team 2025), and gpt-oss alternates global layers with windows of 128 tokens (OpenAI 2025). The global layers restore the $O(1)$ path length. The windowed layers keep both the arithmetic and the memory of generation small: a model that generates text caches the keys and values of past tokens (Section 10.3), and a windowed layer needs to keep only the last w of them. With r windowed layers per global layer, the cache shrinks to

$$\frac{r \min(n, w) + n}{(r+1)\, n}$$

of the cache of a stack of global layers only: each of the r windowed layers caches $\min(n, w)$ positions and the global layer all n, against $(r+1)n$ positions for global layers throughout. For Gemma 3's $r = 5$ and $w = 1024$, this is about a quarter at $n = 8192$, and it approaches $1/6$ as the context grows.

Llama 4 also drops the positional encoding from its global layers, which then work like the NoPE of Section 10.4. It keeps the rotary encoding only in its local layers, which attend

within chunks of the sequence and therefore see only short offsets (Meta AI 2025). The aim is a model that runs at contexts longer than those it was trained on.

10.5.3.3 A Linear-Cost Implementation

The mask defines the result, but by itself it saves no work. A masked softmax over a band still scores all n^2 pairs and then discards most of them. The efficient implementation processes the sequence in blocks of w queries. A query can attend only to keys in its own block and in the block before it, so each block scores a $w \times 2w$ tile. The total work is $2nw$ scores instead of n^2, which is linear in n for fixed w.

```
def windowed_attention(Q, K, V, w):
    """Causal sliding-window attention in O(nw) time and memory."""
    n, d = Q.shape
    assert n % w == 0, 'blocks must tile the sequence exactly'
    Qb, Kb, Vb = (X.reshape(n // w, w, -1) for X in (Q, K, V))
    # Each query block attends to its own key block and the one before it
    prev = lambda X: torch.cat([torch.zeros_like(X[:1]), X[:-1]])
    Kb, Vb = torch.cat([prev(Kb), Kb], 1), torch.cat([prev(Vb), Vb], 1)
    i = torch.arange(w, device=Q.device)[:, None]      # Query in its block
    j = torch.arange(-w, w, device=Q.device)[None, :]  # Key offset from it
    first = torch.arange(n // w, device=Q.device)[:, None, None] == 0
    mask = (j <= i) & (i - j < w) & ~(first & (j < 0))  # No keys before 0
    scores = Qb @ Kb.transpose(-1, -2) / math.sqrt(d)
    return (d2l.masked_softmax(scores, mask) @ Vb).reshape(n, -1)
```

The blocked version should agree with the mask-based definition.

```
n, w = 2048, 128
Q, K, V = (torch.randn(n, d_h, device=d2l.try_gpu()) for _ in range(3))
scores = Q @ K.T / math.sqrt(d_h)
reference = d2l.masked_softmax(scores, window_mask(n, w, Q.device)) @ V
err = (windowed_attention(Q, K, V, w) - reference).abs().max()
print(f'maximum deviation: {float(err):.2e}')
```

```
maximum deviation: 4.17e-07
```

A fused kernel cannot skip single scores. It computes in tiles of $b \times b$ scores held in on-chip memory, and it can skip a tile only if the mask excludes all of it. Tiles inside the mask are computed without masking, and only tiles that straddle its edge apply the mask entry by entry. The cost of a mask in a fused kernel is therefore the fraction of tiles it touches. We count it for tiles of $b = 128$, the usual size.

```
def tiles_visited(mask, b):
    """Fraction of b-by-b tiles of a mask that hold at least one True."""
    T = mask.shape[0] // b
    return mask.reshape(T, b, T, b).any(dim=(1, 3)).float().mean().item()

n, b, w = 2048, 128, 128
print(f'causal:  {tiles_visited(d2l.causal_mask(n), b):.1%} of tiles')
print(f'window:  {tiles_visited(window_mask(n, w), b):.1%} of tiles')
```

```
causal:  53.1% of tiles
window:  12.1% of tiles
```

A causal mask touches just over half of the tiles, and a window of 128 touches about one in eight: the diagonal tile and its left neighbor in each row of tiles. A windowed fused kernel therefore costs time proportional to nw, rounded up to whole tiles, and narrowing the window below the tile size saves nothing more. Fused kernels take the window as part of their interface, and we check that their output equals that of `windowed_attention`:

```
def band_mod(w):
    """FlexAttention mask_mod: True where query q may attend to key k."""
    return lambda batch, head, q, k: (k <= q) & (q - k < w)

block_mask = create_block_mask(band_mod(w), None, None, n, n,
                               device=d2l.try_gpu())
print(f'tiles skipped: {block_mask.sparsity():.1f}%')
Q, K, V = (torch.randn(1, 1, n, d_h, device=d2l.try_gpu()) for _ in range(3))
O = torch.compile(flex_attention)(Q, K, V, block_mask=block_mask)
O_ref = windowed_attention(Q[0, 0], K[0, 0], V[0, 0], w)
print(f'maximum deviation: {float((O[0, 0] - O_ref).abs().max()):.2e}')
```

```
tiles skipped: 87.9%
maximum deviation: 6.56e-07
```

FlexAttention (Dong et al. 2025) makes this tile map the interface to a fused kernel. The user writes the mask as a function of the query and key positions, here the band, and `create_block_mask` evaluates it once per tile. The resulting `BlockMask` is the tile map counted above, and its sparsity is the percentage of tiles skipped. `flex_attention` with this block mask equals `windowed_attention`. Under `torch.compile`, as here, it generates a kernel in the style of FlashAttention that visits only the kept tiles; called without compilation, it would compute the full score matrix. Every mask built from the pieces of Section 10.2 (windows, packed documents, prefixes) then runs at the speed of a fused kernel.

Fixed windows decide by position what may be attended, before the content is seen. Natively sparse attention instead learns which blocks of keys each query visits, at the same block granularity, and runs as a fused kernel that visits only the chosen blocks (Yuan et al. 2025).

10.5.4 Linear Attention in Recurrent Form

10.5.4.1 Kernelizing the Score

The third approach changes the softmax itself. Attention pooling with a fixed kernel (Section 10.1) weights each value in proportion to a similarity kernel, and the exponentiated dot product $\exp(\mathbf{q}^\top \mathbf{k}/\sqrt{d})$ is one kernel among many. Suppose we choose instead a kernel that *factorizes*, $k(\mathbf{q}, \mathbf{k}) = \phi(\mathbf{q})^\top \phi(\mathbf{k})$ for some elementwise feature map ϕ. Causal attention with this kernel is

$$\mathbf{o}_t^\top = \frac{\sum_{s \le t} \phi(\mathbf{q}_t)^\top \phi(\mathbf{k}_s)\, \mathbf{v}_s^\top}{\sum_{s \le t} \phi(\mathbf{q}_t)^\top \phi(\mathbf{k}_s)} = \frac{\phi(\mathbf{q}_t)^\top \mathbf{S}_t}{\phi(\mathbf{q}_t)^\top \mathbf{z}_t}, \qquad \mathbf{S}_t = \sum_{s \le t} \phi(\mathbf{k}_s)\mathbf{v}_s^\top, \quad \mathbf{z}_t = \sum_{s \le t} \phi(\mathbf{k}_s). \tag{10.30}$$

Because the query enters linearly, it factors out of the sum. The remaining terms, $\mathbf{S}_t$ and $\mathbf{z}_t$, do not depend on the query. Reordering the sums therefore removes the explicit pairwise interaction and its quadratic cost. Instead of t scores, each query needs one matrix-vector product against a running summary (Katharopoulos et al. 2020). Following that paper, we take $\phi(x) = \mathrm{elu}(x) + 1$, which keeps features positive so that the normalizer cannot vanish.

The sums in Equation 10.30 satisfy the recurrences

$$\mathbf{S}_t = \mathbf{S}_{t-1} + \phi(\mathbf{k}_t)\,\mathbf{v}_t^\top, \qquad \mathbf{z}_t = \mathbf{z}_{t-1} + \phi(\mathbf{k}_t), \tag{10.31}$$

which define a fixed-size state of $d \times d_v$ numbers. The state is updated by an outer-product write as each token arrives, then read by the query. Linear attention is a recurrent network whose hidden state is a matrix. The state acts as an associative memory, with $\phi(\mathbf{k}_t)$ the address and $\mathbf{v}_t$ the content. Schlag et al. (2021) showed that this layer is the fast weight programmer of the early 1990s. The same computation has two implementations. The *parallel* form for training computes all states at once with cumulative sums. The *recurrent* form for generation carries $(\mathbf{S}, \mathbf{z})$ forward one token at a time, in constant memory, like every RNN in Chapter 8.

```
def phi(X):
    return F.elu(X) + 1

def linear_attention_parallel(Q, K, V):
    """Training form: all outputs at once via cumulative sums."""
    Qp, Kp = phi(Q), phi(K)
    S = torch.cumsum(Kp[:, :, None] * V[:, None, :], dim=0)  # (n, d, d_v)
    z = torch.cumsum(Kp, dim=0)                              # (n, d)
    return (Qp[:, :, None] * S).sum(1) / (Qp * z).sum(-1, keepdim=True)

def linear_attention_recurrent(Q, K, V):
    """Generation form: a (d, d_v) state, updated token by token."""
    Qp, Kp = phi(Q), phi(K)
    S = torch.zeros(Q.shape[-1], V.shape[-1], device=Q.device)
    z = torch.zeros(Q.shape[-1], device=Q.device)
    outputs = []
    for t in range(Q.shape[0]):
        S = S + Kp[t][:, None] * V[t][None, :]   # Outer-product write
        z = z + Kp[t]
        outputs.append(Qp[t] @ S / (Qp[t] @ z))  # Query the memory
    return torch.stack(outputs)
```

Equations Equation 10.31 and Equation 10.30 sum the same terms in different orders, so their implementations should agree up to floating-point rounding.

```
n = 512
Q, K, V = (torch.randn(n, d_h, device=d2l.try_gpu()) for _ in range(3))
err = (linear_attention_parallel(Q, K, V)
       - linear_attention_recurrent(Q, K, V)).abs().max()
print(f'maximum deviation: {float(err):.2e}')
print(f'recurrent state: {d_h}x{d_h} + {d_h} floats '
      f'= {(d_h * d_h + d_h) * 4 / 1024:.0f} KiB at any sequence length')
```

```
maximum deviation: 4.77e-07
recurrent state: 64x64 + 64 floats = 16 KiB at any sequence length
```

At generation time this linear-attention layer with $d_h = 64$ carries 16 KiB of state, whatever the context length. A softmax head instead caches every past key and value, $2nd_h \cdot 4$ bytes in fp32, or about 4 MiB at $n = 8192$. Its cache grows linearly with the generated sequence, whereas the recurrent state stays fixed.

The state $\mathbf{S}_t$ is a lossy summary. It holds $d \times d_v$ numbers however many tokens it has absorbed, so as the context grows the writes interfere, and retrieving one particular value becomes approximate. Softmax attention keeps every key and can put nearly all of a query's weight on one of them. A kernel with positive features such as elu$+1$ cannot become nearly one-hot in the same way: scaling up a query sharpens the softmax without limit but leaves the normalized weights of this kernel almost unchanged (Exercise 6).

10.5.4.2 Chunked Computation

Neither form suits a GPU during training. The parallel form stores the state at every position, n matrices of $d \times d_v$ numbers: 256 MiB for one head at $n = 16{,}384$ in fp32, more than the score matrix it replaces until n exceeds $d\,d_v = 4096$. The recurrent form stores one state but takes n sequential steps of matrix–vector work, which leaves the matrix units of a GPU idle. Practical kernels divide the sequence into chunks of c positions, as online softmax divided the keys (Yang, Wang, Shen, et al. 2024). Inside a chunk, the outputs come from the quadratic form, a causally masked $c \times c$ matrix of kernel values. Across chunks, the state carries the sum over all earlier chunks. For a chunk that starts at position t_0, with $\mathbf{S}$ and $\mathbf{z}$ the sums over all positions before t_0,

$$\mathbf{o}_t^\top = \frac{\phi(\mathbf{q}_t)^\top \mathbf{S} + \sum_{t_0 \le s \le t} \phi(\mathbf{q}_t)^\top \phi(\mathbf{k}_s)\, \mathbf{v}_s^\top}{\phi(\mathbf{q}_t)^\top \mathbf{z} + \sum_{t_0 \le s \le t} \phi(\mathbf{q}_t)^\top \phi(\mathbf{k}_s)}. \tag{10.32}$$

The first term of the numerator reads the memory of the earlier chunks, and the second attends within the chunk. Both are matrix products. For each chunk, a $c \times d$ by $d \times c$ product gives the kernel values, and the state advances by a $d \times c$ by $c \times d_v$ product. The total work is $O(ncd + nd\,d_v)$, linear in n, and only one state per chunk is stored.

```python
def linear_attention_chunked(Q, K, V, c=64):
    """Training form in chunks: quadratic within, recurrent across."""
    n, d = Q.shape
    assert n % c == 0, 'chunks must tile the sequence exactly'
    Qp, Kp = phi(Q).reshape(n // c, c, d), phi(K).reshape(n // c, c, d)
    Vc = V.reshape(n // c, c, -1)
    # The state each chunk starts from: sums over all earlier chunks
    S = torch.cumsum(Kp.transpose(-1, -2) @ Vc, dim=0)   # (n/c, d, d_v)
    z = torch.cumsum(Kp.sum(1), dim=0)                   # (n/c, d)
    S = torch.cat([torch.zeros_like(S[:1]), S[:-1]])
    z = torch.cat([torch.zeros_like(z[:1]), z[:-1]])
    A = torch.tril(Qp @ Kp.transpose(-1, -2))            # (n/c, c, c)
    num = Qp @ S + A @ Vc                                # Past + this chunk
    den = Qp @ z[..., None] + A.sum(-1, keepdim=True)
    return (num / den).reshape(n, -1)
```

The chunked form sums the same terms once more, so it too should match the recurrent form.

```
n = 512
Q, K, V = (torch.randn(n, d_h, device=d2l.try_gpu()) for _ in range(3))
err = (linear_attention_chunked(Q, K, V)
       - linear_attention_recurrent(Q, K, V)).abs().max()
print(f'maximum deviation: {float(err):.2e}')
```

```
maximum deviation: 3.58e-07
```

The three forms agree to floating-point rounding. They differ only in what they keep in memory and in how much of the work is matrix products.

10.5.4.3 Linear Attention in Practice

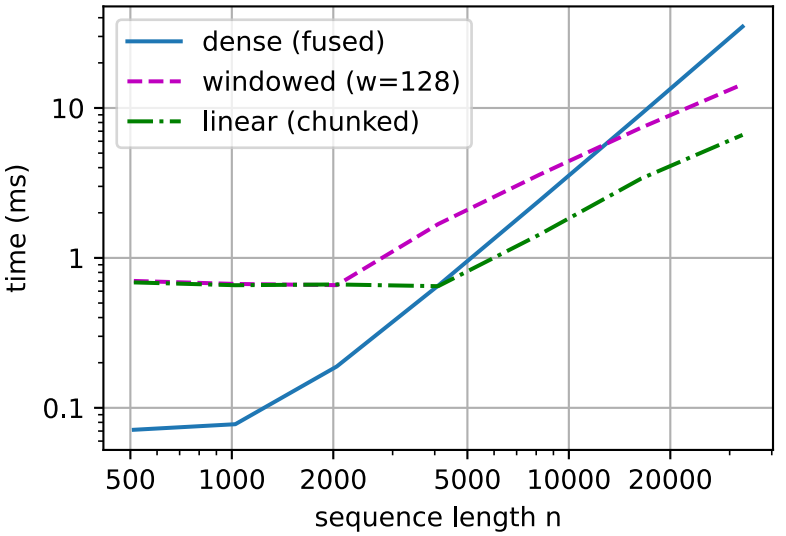
Plain kernelized attention of this kind did not replace softmax attention in language models. At equal size it models language less well, and it is weakest on tasks that must recall one particular earlier token (Arora, Eyuboglu, Zhang, et al. 2024; Yang, Wang, Shen, et al. 2024). Two changes made the recurrent form competitive. First, the state learns to forget: a decay computed from each input multiplies $\mathbf{S}_{t-1}$ before the new write, so that old content fades when new content needs the room (Exercise 10 works out the case of a constant decay). Second, models keep a few softmax layers for recall. MiniMax-01, for instance, places one softmax attention layer after every seven linear-attention layers (MiniMax 2025). A recurrence with a fixed-size state that is updated linearly, as in Equation 10.31, is a *linear state space model*. Section 12.2 develops these models and their input-dependent decay from this starting point, and Section 12.4 extends `linear_attention_chunked` to a state that decays.

10.5.5 Measured Running Time

We now time the three strategies of this section on equal terms, as the sequence grows from 512 to 32,768 tokens: dense causal attention in the fused kernel, windowed attention with $w = 128$, and chunked linear attention. All three process 32 heads of width 64 at once, for instance a minibatch of 4 sequences with 8 heads each. They run in bfloat16, the precision in which attention is trained, and `vmap` maps the one-head functions of this section over the heads. Enough work per call keeps the fixed cost of launching kernels from hiding the growth with n. The milliseconds depend on the GPU, but the slopes on the log–log plot follow from the operation counts.

```
H, lengths = 32, [512, 1024, 2048, 4096, 8192, 16384, 32768]
mechanisms = {
    'dense (fused)': lambda Q, K, V: fused_attention(Q[None], K[None],
                                                     V[None]),
    'windowed (w=128)': torch.vmap(
        lambda Q, K, V: windowed_attention(Q, K, V, 128)),
    'linear (chunked)': torch.vmap(linear_attention_chunked)}
times = {name: [] for name in mechanisms}
for n in lengths:
    X = torch.randn(H, n, d_h, device=d2l.try_gpu(), dtype=torch.bfloat16)
    for name, f in mechanisms.items():
```

```
        times[name].append(d2l.Benchmark(lambda: f(X, X, X)).time * 1e3)
d2l.plot(lengths, list(times.values()), 'sequence length n', 'time (ms)',
         legend=list(times), xscale='log', yscale='log', figsize=(4.5, 3))
```



At short lengths all three take well under a millisecond, and fixed per-call costs rather than the operation counts decide which is fastest. At long lengths the time of dense attention grows about fourfold with every doubling of n, while the times of the windowed and the chunked linear mechanisms grow about twofold. At 32,768 tokens each of them therefore needs well under half the time of dense attention, and the gap widens with every further doubling. Dense attention remains the default at moderate lengths, where it is exact and fast. At long contexts the mechanisms of linear cost are faster, and linear attention is also faster during generation, where its state stays the size of one matrix.

Run eagerly, the windowed and the linear functions launch many small kernels, one per operation, while dense attention is a single fused kernel. Below a few thousand tokens these launches cost more than the arithmetic, so the dense kernel is the fastest there and the two others overtake it only as n grows.

10.5.6 Summary

Dense attention costs $4n^2d$ FLOPs and, in a naive implementation, two $n \times n$ buffers per head. Online softmax computes the same result one key block at a time while carrying a running maximum, normalizer, and output, and FlashAttention fits this schedule to the memory hierarchy, so exact attention runs in linear memory and several times faster. Sliding windows reduce the work to $O(nw)$ and widen the receptive field to $1 + L(w-1)$ after L layers; interleaved global layers restore direct paths, and a fused kernel saves work only in whole tiles. A factorizing kernel turns attention into a recurrent network with a $d \times d_v$ matrix state, computed in chunks for training and step by step for generation; the state has constant size but is a lossy summary of the past. Measured, dense time grows fourfold per doubling of the length and the time of the linear-cost mechanisms twofold, so at long contexts they need a fraction of its time.

10.5.7 Exercises

1. **Reaching across the sequence.** Table 10.1 compares layers one at a time, but connecting two distant tokens may take many layers. Count one multiply–add as two FLOPs, so that a convolution layer with kernel k costs $2knd^2$, an RNN layer $4nd^2$, and a self-attention layer $8nd^2 + 4n^2d$. Let $n = 4096$, $d = 512$, and $k = 3$.
   1. A stack of plain convolutions must connect the first token with the last. Find the number of layers it needs and its total FLOPs, show that this total grows quadratically in n, as the cost of one attention layer does, and compare the two totals numerically.
   2. Dilated convolutions (Section 6.3.3) with dilation k^{l-1} in layer l grow the receptive field geometrically. Find the receptive field after L layers, the depth needed to connect the two tokens, and the total FLOPs of that stack.
   3. Rank the four designs, namely one attention layer, the two convolution stacks, and one RNN layer, by total FLOPs and by sequential steps, counting each layer of a stack as one step and each token of the RNN as one step. Which resource does each design spend to connect the two tokens?
2. code **The running maximum.** Remove the running maximum from `chunked_attention`, so that the state accumulates plain sums of e^{a_j}. Run the result with **Q**, **K**, **V** and the state s, **O** in float16, on queries scaled up by a factor of 10.
   1. Before running, compute the largest exponent x for which e^x is finite in float16, and estimate the fraction of scaled scores that exceed it when queries and keys of width 64 have independent standard normal entries.
   2. Run the modified function, and report at which operation the first infinite or undefined value appears.
   3. Explain why carrying m bounds every exponent by zero. A term $e^{a_j - m'}$ that is smaller than the smallest positive float16 number rounds to zero. Bound the total contribution of such terms to s, relative to the term of the largest score, for a row of n keys.
   4. `chunked_attention` initializes m with the most negative finite value of the dtype and masks scores with the same value, the idiom of Section 10.2. Take $n = 2c$ for blocks of c keys, and let the loop visit the two key blocks in reverse order, so that every score of the first visited block is masked for a query in the first block. Trace m, s, and **O** for that query through both iterations, and show that the final output is still exact.
3. **Merging partial states.** Fix one query with scores a_j and values $\mathbf{v}_j$. For a set B of keys, call the triple $(m_B, s_B, \mathbf{o}_B)$ that Equation 10.29 produces from the keys in B alone the partial state of B, so that $\mathbf{o}_B/s_B$ is the attention output restricted to B.
   1. Derive the partial state of $B \cup B'$ for disjoint sets B and B' from the two partial states, and show that Equation 10.29 is the special case in which B' is a single block.
   2. Show that this merge is associative and commutative. Conclude that the key blocks can be processed in any order, or in parallel and merged in a tree of depth $\lceil \log_2(n/c) \rceil$ for blocks of c keys, and explain why this matters when a single query attends to a long cache during generation.
   3. Replace the pair (m_B, s_B) by the single number $\ell_B = m_B + \log s_B$ and the triple by $(\ell_B, \bar{\mathbf{o}}_B)$ with $\bar{\mathbf{o}}_B = \mathbf{o}_B/s_B$. Express the merged ℓ and $\bar{\mathbf{o}}$ in these terms, and show that $\bar{\mathbf{o}}$ is a convex combination of $\bar{\mathbf{o}}_B$ and $\bar{\mathbf{o}}_{B'}$ whose weights depend only on ℓ_B and $\ell_{B'}$.
4. **Arithmetic intensity of attention.** The count Equation 10.28 extends to values of

width d_v: for n queries and keys of width d, the two matrix products of Equation 10.12 cost $2n^2d$ and $2n^2d_v$ floating-point operations, and the softmax costs a few operations per score.

1. Suppose that the score matrix is written to memory once and read once by the softmax, whose output is written once and read once by the value product, all in single precision. Count the bytes moved by these four passes, and by reading **Q**, **K**, **V** and writing the output.
2. Neglecting the softmax arithmetic, form the ratio of operations to bytes and let $n \to \infty$ at fixed d and d_v. Show that the limit is $(d + d_v)/8$, and state how the softmax arithmetic changes it.
3. Look up the peak arithmetic throughput and the memory bandwidth of one accelerator, compute its ratio of operations per byte, and decide whether attention with $d = d_v = 64$ is limited by arithmetic or by memory traffic on it.

5. code **Tiles of a fused kernel.** A fused kernel works on tiles of $b \times b$ scores and skips a tile only if the mask excludes all of it. Let $T = n/b$.
   1. Show that a causal mask leaves $T(T+1)/2$ of the T^2 tiles, and that a window of width w, a multiple of b, leaves about $T(w/b + 1)$. Check both counts with `tiles_visited`. Below which width does narrowing the window stop saving tiles?
   2. Several documents are packed into one sequence of $n = 2048$ tokens, and the mask combines causality with "same document" as in Section 10.2, using `tiles_visited` and $b = 128$. Compare four documents of 512 tokens with documents of 1,000, 600, 300, and 148 tokens. Explain why documents whose boundaries do not fall on tile boundaries cost more tiles.
   3. A kernel that receives the mask as an $n \times n$ boolean tensor must read it from memory. Compare the size of that tensor at $n = 32{,}768$ with the size of **Q**, **K**, and **V** for 8 heads of width 64 in bfloat16, and explain why FlexAttention evaluates the mask as a rule, once per tile.
6. code **The feature map.** The feature map $\phi(x) = \text{elu}(x) + 1$ in `phi` is one choice among many. Compare it with $\phi(x) = \text{relu}(x)$ and $\phi(x) = \exp(x)$, both applied elementwise, on $n = 512$ random queries and keys of width `d_h`.
   1. For each map, compute the causal linear-attention weight matrix with entries

$$\frac{\phi(\mathbf{q}_t)^\top \phi(\mathbf{k}_s)}{\sum_{s' \le t} \phi(\mathbf{q}_t)^\top \phi(\mathbf{k}_{s'})}, \qquad s \le t,$$

      and measure its maximum absolute difference from the causal softmax weight matrix, which `causal_attention` forms before multiplying by **V**, and the mean entropy of its rows.
   2. Under which condition on $\mathbf{q}_t$ and the keys $\mathbf{k}_1, \dots, \mathbf{k}_t$ does the ReLU map produce an undefined row? Derive the probability of this event for row t when all entries are independent standard normal, and evaluate it for $t = 1$ at width 64 and at width 4.
   3. Multiply the queries by $\beta \in \{1, 4, 16\}$, and recompute the mean row entropy for the three maps and for the softmax weights. Explain from a property of each ϕ which maps cannot sharpen their weights in this way, and what this implies for approximating a softmax row that is nearly one-hot.
7. **Traffic of a tiled schedule.** ⊠ By Exercise 4a, the naive layer moves $4n^2$ floats to and from off-chip memory beyond the terms linear in n, since the score matrix and the weight matrix are each written once and read once. Ignore all traffic that is linear in n.

1. Suppose that `chunked_attention` runs one operation at a time, each reading its inputs from off-chip memory and writing its output back. Count, as functions of n, c, and d, the floats moved for the score stripes, the floats moved for $\mathbf{Q}$ and the accumulator $\mathbf{O}$, and the number of operations launched. Which of these quantities does chunking reduce relative to the naive layer?
2. A fused kernel instead holds a tile of b queries, their output accumulator, and one tile each of b keys and b values in an on-chip memory of M floats, so that $4bd \leq M$; scores and per-row statistics stay in registers. It loads each query tile once, streams all key and value tiles past it, and writes each output tile once. Show that its traffic beyond the linear terms is $2n^2d/b$ floats and that the largest admissible tile gives $\Theta(n^2d^2/M)$, the bound that Dao et al. (2022) prove for FlashAttention.
3. Compute the ratio of the fused traffic to the naive traffic for $d \in \{64, 128, 256\}$ with $M = 2^{16}$, and state how the head width limits the benefit of tiling.

8. **An exact feature map.** ▣ The softmax kernel $\kappa(\mathbf{q}, \mathbf{k}) = \exp(\mathbf{q}^\top \mathbf{k}/\sqrt{d})$ also factorizes, but only with infinitely many features.
   1. Expand κ in its Taylor series, and show that the degree-p term equals $\phi_p(\mathbf{q})^\top \phi_p(\mathbf{k})$ for a feature map ϕ_p of dimension d^p built from the p-fold tensor product of its argument.
   2. Truncate the series after degree 2. Show that the truncated kernel is positive for every pair of $\mathbf{q}$ and $\mathbf{k}$, so the normalizer of Equation 10.30 cannot vanish, and give the dimension of the feature map that concatenates ϕ_0, ϕ_1, and ϕ_2. For $d = 64$ and values of width 64, compute the size of the state $(\mathbf{S}, \mathbf{z})$ in single precision and the length n at which the key–value cache of a softmax head reaches the same size. A truncation of this kind is the feature map of Based (Arora, Eyuboglu, Zhang, et al. 2024).
   3. Take as given that the $n \times n$ matrix with entries $\kappa(\mathbf{x}_i, \mathbf{x}_j)$ is positive definite for distinct $\mathbf{x}_1, \ldots, \mathbf{x}_n$. Show that no feature map of dimension $D < n$ reproduces κ exactly on these points when queries and keys coincide.
9. [extended] **Attention variants in a language model.** ▣ Whether the quality gap between exact and linear attention matters depends on the task. Measure it on `TinyCharLM` of Section 10.4, with `pos='learned'` for every variant so that position enters the same way.
   1. Subclass `TinyCharLM` and override `attend` twice: once with a causal band mask of width w in place of the causal mask, and once with the causal linear-attention weights of Equation 10.30 for $\phi(x) = \mathrm{elu}(x) + 1$, computed explicitly as a $T \times T$ matrix so that the method still returns its weights.
   2. Train dense attention, windows of width $w \in \{4, 16, 64\}$, and linear attention with the protocol of Section 10.4.4.3, and report validation perplexity at context 128. Relate the window results to the reach $1 + 2(w - 1)$ of the two-block model.
   3. Build validation sequences of 128 bytes, each a random 64-byte excerpt of the validation text followed by the same excerpt again, and compare the mean loss on the second half with that on the first half for every variant. Which variants exploit the repetition, and how does the answer relate to the reach of the windowed models and to the fixed-size state of linear attention?
10. code **A decaying state.** ▣ Give the state of Equation 10.31 a scalar decay $\gamma \in (0, 1)$,

$$\mathbf{S}_t = \gamma\, \mathbf{S}_{t-1} + \phi(\mathbf{k}_t)\, \mathbf{v}_t^\top, \qquad \mathbf{z}_t = \gamma\, \mathbf{z}_{t-1} + \phi(\mathbf{k}_t). \tag{10.33}$$

    1. Show that the parallel form now weights the contribution of position s to the output at t by γ^{t-s}, and implement it by rescaling the cumulative sums of `linear_attention_parallel` with powers of γ.

2. Add the decay to `linear_attention_recurrent`, and verify that the two forms agree for $n = 512$ and $\gamma = 0.99$.
3. The rescaling multiplies position s by γ^{-s} before the cumulative sum. For which n does this overflow single precision at $\gamma = 0.9$ and at $\gamma = 0.5$?
4. Add the decay to `linear_attention_chunked`: rescale each position relative to the start of its chunk, and carry the state from one chunk to the next with the decayed recurrence. Show that no factor larger than γ^{-c} appears, run the result for $\gamma = 0.5$, $n = 512$, and $c = 64$, and compare it with the recurrent form.
5. Compute the lag at which the weight γ^{t-s} falls below 0.01 for $\gamma = 0.99$. A task requires recalling one token from 2000 positions back, while the tokens in between are filler. Explain why no fixed γ serves both that token and the filler, and what a decay computed from each token's own input could do instead.

10.6 Attention Circuits

Since Section 10.1 we have plotted attention weights as heatmaps, and that section warned that such a map is a hypothesis about what a layer does, not a proof. A map shows where each query looks, and it invites a simple reading: the model predicted *B* because it attended to *B*. The map shows only half of a head. What the head adds to its output also depends on its value and output projections, so a head that attends sharply to a token may pass on the token's identity, one feature of it, or nothing at all. Explaining what a trained model computes therefore takes both halves of every head, and experiments that tell a mechanism apart from a pattern that merely looks like one.

This section works through one complete case. In a transformer built from attention layers alone, each head factors into two matrices that can be read from its weights: a *QK circuit* that decides where the head attends and an *OV circuit* that decides what it moves (Elhage et al. 2021). Expanding the layers algebraically shows which functions one and two layers can express. We then train the attention-only `TinyCharLM` of Section 10.4 on sequences that repeat a random pattern, find the two-layer *induction head* that completes the repetition, and test it four ways: through the loss, the attention maps, the weights, and the removal of heads. The same circuit appears in large language models, where it has been tied to their ability to learn from the context (Olsson et al. 2022).

```
import copy
```

10.6.1 The Residual Stream

10.6.1.1 Residual Representations Across Layers

Reading a head from its weights requires an exact account of what each layer adds to a token's representation. We write it down for the model we will train. `TinyCharLM` has the simplest architecture that still counts as a transformer. Each of its L blocks adds the output of H causal attention heads to its input, and nothing else: there are no feed-forward layers and no normalization. The output head reuses the embedding matrix $\mathbf{E}$, whose rows are the token embeddings. Write $\mathbf{e}_{x_i} \in \mathbb{R}^d$ for the embedding of token x_i and set $\mathbf{h}_i^{(0)} = \mathbf{e}_{x_i}$. Block ℓ computes, for every position i,

$$\mathbf{h}_i^{(\ell)} = \mathbf{h}_i^{(\ell-1)} + \sum_{m=1}^{H} \sum_{j \le i} \alpha_{ij}^{\ell m} \, \mathbf{W}_{\mathrm{OV}}^{\ell m} \, \mathbf{h}_j^{(\ell-1)}, \tag{10.34}$$

where $\alpha_{ij}^{\ell m}$ is the weight with which head m of block ℓ attends from position i to position j, and $\mathbf{W}_{\mathrm{OV}}^{\ell m}$ is a $d \times d$ matrix, defined below, that combines the head's value and output projections. The output head scores the final vector against every embedding, $\mathbf{z}_i = \mathbf{E}\,\mathbf{h}_i^{(L)}$. The sequence $\mathbf{h}_i^{(0)}, \ldots, \mathbf{h}_i^{(L)}$ is the *residual stream* of position i (Figure 10.12). Each block adds to this vector rather than replacing it. A head reads the streams at earlier positions, transforms what it reads linearly, and adds the result to the stream at the current position. Information that one block writes therefore stays available to every later block and to the output head.

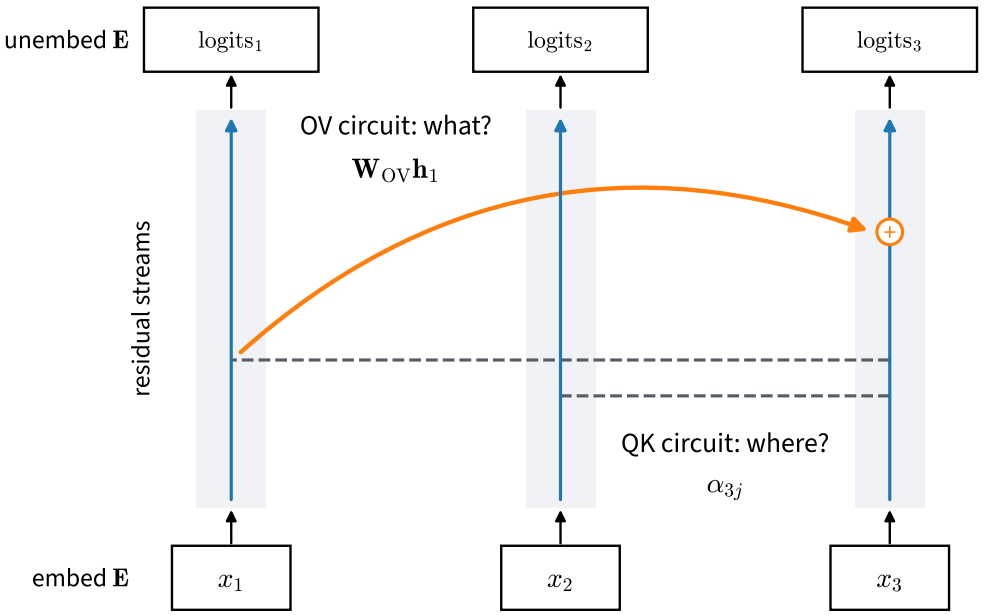


Figure 10.12: The residual stream. Each position carries a vector from its embedding to the output logits. The QK circuit determines the attention weights α_{3j}, and the OV circuit determines the contribution of each attended vector to the destination stream.

Once the attention weights α are fixed, the map from embeddings to logits in Equation 10.34 is linear, and expanding the recursion writes it as a sum of matrix products along paths through the network. This decomposition is exact for attention-only transformers (Elhage et al. 2021), which is why `TinyCharLM` has no feed-forward layers or normalization. Full transformers have both, and claims about them need the experiments developed below, not algebra alone.

10.6.1.2 QK and OV Circuits

Each head's term in Equation 10.34 splits into two parts. Its attention weights come from a softmax over scores that involve the query and key projections only through their product,

and what it adds depends only on the value and output projections:

$$s_{ij} = \frac{\mathbf{h}_i^\top \mathbf{W}_{\mathrm{QK}} \mathbf{h}_j}{\sqrt{d_h}}, \qquad \mathbf{W}_{\mathrm{QK}} = \mathbf{W}_q^\top \mathbf{W}_k, \qquad \mathbf{W}_{\mathrm{OV}} = \mathbf{W}_o \mathbf{W}_v. \tag{10.35}$$

Vectors are columns here: $\mathbf{W}_q, \mathbf{W}_k, \mathbf{W}_v \in \mathbb{R}^{d_h \times d}$ map the stream into a head of width d_h, and $\mathbf{W}_o \in \mathbb{R}^{d \times d_h}$ maps the head's output back. The *QK circuit* $\mathbf{W}_{\mathrm{QK}}$ is a bilinear form that scores a query position against a key position. The *OV circuit* $\mathbf{W}_{\mathrm{OV}}$ maps an attended stream vector to what the head writes. The products, not the four projections, are the right objects to study, because the projections are not identifiable. Without rotary embeddings, replacing $\mathbf{W}_q$ by $\mathbf{R}\mathbf{W}_q$ and $\mathbf{W}_k$ by $\mathbf{R}^{-\top}\mathbf{W}_k$, for any invertible $d_h \times d_h$ matrix $\mathbf{R}$, changes nothing the model computes and leaves $\mathbf{W}_{\mathrm{QK}}$ as it was. With rotary embeddings the same holds for every $\mathbf{R}$ that commutes with the rotations, such as a rotation and scaling of each frequency pair.

Both circuits are $d \times d$ matrices of rank at most $d_h = d/H$, so a head reads and writes only a d_h-dimensional slice of the stream. With $d = 128$ and four heads, each head works through a rank-32 bottleneck. In `TinyCharLM` the four projections live in two fused layers: `qkv` stacks the query, key, and value maps, and `proj` holds the output maps of the heads side by side. Neither layer has a bias by default, so reshaping the two weight matrices to expose a head axis gives the circuits of every head at once. The function `circuits` forms them for one block, and the cell below prints those of head 0 in block 1 of a freshly initialized model.

```
def circuits(model, blk):
    """W_QK and W_OV of every head of block blk, each (heads, D, D)."""
    W_qkv = model.blks[blk]['qkv'].weight           # (3D, D): W_q; W_k; W_v
    W_o = model.blks[blk]['proj'].weight            # (D, D): heads in columns
    H, D = model.num_heads, W_o.shape[0]
    W_q, W_k, W_v = W_qkv.reshape(3, H, D // H, D)  # Each (H, d_h, D)
    W_QK = W_q.mT @ W_k
    W_OV = W_o.reshape(D, H, D // H).transpose(0, 1) @ W_v
    return W_QK, W_OV

model = d2l.TinyCharLM(vocab_size=64, pos='rope')
W_QK, W_OV = circuits(model, 0)
for name, M in (('W_QK', W_QK[0]), ('W_OV', W_OV[0])):
    print(f'{name}: shape {tuple(M.shape)}, '
          f'rank {torch.linalg.matrix_rank(M).item()}')
```

```
W_QK: shape (128, 128), rank 32
W_OV: shape (128, 128), rank 32
```

`nn.Linear` stores its weight with shape (output, input), the orientation of the column-vector equations above. The rows of `qkv.weight` therefore stack $\mathbf{W}_q$, $\mathbf{W}_k$, and $\mathbf{W}_v$, and the columns of `proj.weight` split by head.

Both circuits are 128×128 and have rank 32, the head width. Rotary embeddings add position to the QK circuit and nowhere else. RoPE rotates the query at position i and the key at position j by angles proportional to their positions, so by Equation 10.23 the score becomes $\mathbf{h}_i^\top \mathbf{W}_q^\top \mathbf{R}_{j-i} \mathbf{W}_k \mathbf{h}_j / \sqrt{d_h}$, with the relative rotation $\mathbf{R}_{j-i}$ between the two projections. The values are not rotated, so the OV circuit does not depend on position. We use `pos='rope'`

throughout because rotary embeddings are the positional scheme of most current language models, including the GPT of Section 11.2.

10.6.1.3 Bigrams, Skip-Trigrams, and the Limits of One Layer

Expanding the residual stream shows how the class of functions a model can represent grows with depth. With *zero* blocks, $\mathbf{z}_i = \mathbf{E}\,\mathbf{e}_{x_i}$: the logits depend on the current token only, which makes the model a bigram model of a constrained kind. Because the output head is tied to the embedding, the logit of token b after token a is $\mathbf{e}_b^\top \mathbf{e}_a$. These logits form the Gram matrix of the embeddings, which is symmetric and of rank at most d, not an arbitrary $|\mathcal{V}| \times |\mathcal{V}|$ lookup table. Training on text fills it with as much of the bigram statistics as that form can hold. With *one* block,

$$\mathbf{z}_i = \mathbf{E}\,\mathbf{e}_{x_i} + \sum_{m=1}^{H} \sum_{j \le i} \alpha_{ij}^{m}\; \mathbf{E}\,\mathbf{W}_{\mathrm{OV}}^{m}\,\mathbf{e}_{x_j}. \tag{10.36}$$

Each new term reads as an if–then rule. If the QK circuit sends attention from the current token to an earlier token x_j, the OV circuit adds $\mathbf{E}\,\mathbf{W}_{\mathrm{OV}}\,\mathbf{e}_{x_j}$ to the logits. Rules of this shape, "when [A] appears somewhere before [B], boost token [C]", are called *skip-trigrams*, and they are all that one layer adds to bigrams. A useful special case is *copying*: attend to earlier occurrences of the current token and boost that same token's logit, so that a word that has appeared becomes more likely to appear again.

One task needs more. Suppose the sequence contains [A][B] … [A], and at the second [A] the model should predict [B], continuing the pattern by looking up what followed last time. No skip-trigram expresses this. The head would have to attend from the second [A] to the token *after* the first one, but in Equation 10.36 the score α_{ij} sees only $\mathbf{e}_{x_i}$, $\mathbf{e}_{x_j}$, and the offset. The key at position j carries no information about its neighbor at $j-1$. A second layer can supply that information. A head in layer 1 attends to the previous token and writes its identity into the stream, so the key at [B]'s position now also says "follows [A]". A head in layer 2 matches the query [A] against these enriched keys, attends to position j, and its OV circuit copies x_j = [B] into the destination stream (Figure 10.13). This two-step circuit is the *induction head*: a previous-token head composed with a match-and-copy head through the residual stream. The argument shows that two layers can express induction and one layer cannot, apart from a position-only shortcut we will meet shortly. Whether gradient descent finds the circuit is a separate, empirical question, and the rest of the section answers it by experiment.

10.6.2 Copying and Induction

10.6.2.1 A Repeated-Sequence Task

To see whether a model learns induction, we need a task that only induction solves. Real text mixes induction with memorized statistics, so we train on sequences of *repeated random tokens* instead: draw a pattern of tokens uniformly from a vocabulary of 64, then tile it until the sequence is full. Within one sequence the pattern repeats; across sequences nothing does, and every bigram is equally likely on average. The first pass over a pattern is therefore unpredictable, while every later pass is predictable from the context alone. The gap between the loss on the first copy and the loss on later copies measures how much the

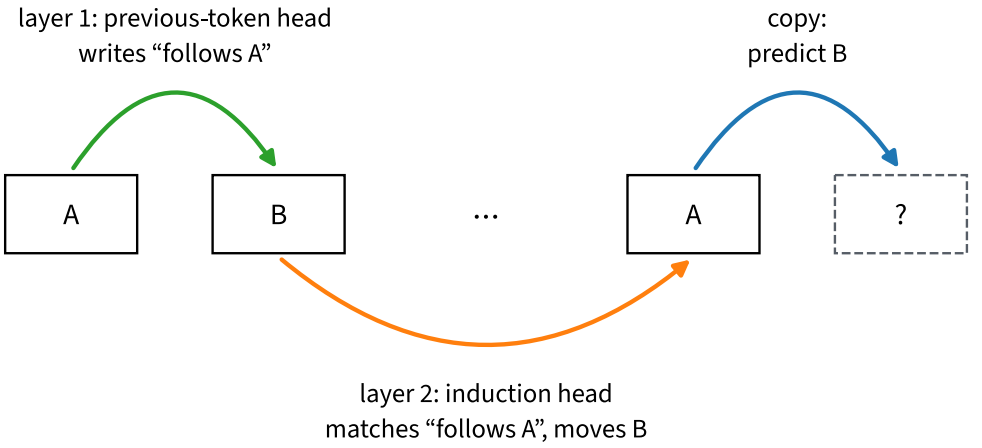


Figure 10.13: The induction circuit spans two layers. In layer 1 a previous-token head writes each token’s predecessor into its stream, so the key at B’s position can announce “follows A”. In layer 2 the induction head’s QK circuit matches the current A against that announcement, and its OV circuit copies B into the prediction.

model retrieves from its context. The function `repeated_batch` draws a minibatch of such sequences, with pattern lengths uniform between `Lmin` and `Lmax`.

```
def repeated_batch(batch_size, num_steps, vocab, Lmin, Lmax, device=None):
    """Random patterns of length Lmin..Lmax, tiled to num_steps tokens."""
    L = torch.randint(Lmin, Lmax + 1, (batch_size,), device=device)
    s = torch.randint(0, vocab, (batch_size, num_steps), device=device)
    pos = torch.arange(num_steps, device=device)
    return torch.gather(s, 1, pos[None, :] % L[:, None]), L

x, L = repeated_batch(batch_size=2, num_steps=16, vocab=10, Lmin=5, Lmax=8)
print(f'pattern lengths: {L.tolist()}')
print(x)
```

```
pattern lengths: [8, 8]
tensor([[4, 0, 7, 6, 0, 2, 5, 2, 4, 0, 7, 6, 0, 2, 5, 2],
        [5, 6, 5, 0, 5, 9, 4, 9, 5, 6, 5, 0, 5, 9, 4, 9]])
```

Each row repeats its first L tokens. Training uses `d2l.train_steps` from Section 9.5, which takes a fixed number of optimizer steps on the mean next-token cross-entropy, here with Adam at learning rate 10^{-3}. A small data module supplies it with an endless stream of fresh 64-token minibatches, so the model never sees the same pattern twice. Every 50 steps, training pauses for a measurement on a fresh batch: `eval_copy` returns the loss on the first copy (targets before position L), the loss on later copies, and the accuracy on later copies.

```
@torch.inference_mode()
def eval_copy(model, x, L):
```

```
    """First-copy loss, later-copy loss, and later-copy accuracy."""
    logits, Y = model(x[:, :-1]), x[:, 1:]
    ce = F.cross_entropy(logits.transpose(1, 2), Y, reduction='none')
    later = torch.arange(1, Y.shape[1] + 1, device=Y.device) >= L[:, None]
    acc = (logits.argmax(-1) == Y)[later].float().mean()
    return torch.stack((ce[~later].mean(), ce[later].mean(), acc)).tolist()

class RepeatedTokens(d2l.DataModule):
    """An endless stream of fresh repeated-token minibatches."""
    def __init__(self, Lmin, Lmax, batch_size=128, num_steps=64, vocab=64):
        super().__init__()
        self.save_hyperparameters()

    def train_dataloader(self):
        while True:
            x, _ = repeated_batch(self.batch_size, self.num_steps,
                                  self.vocab, self.Lmin, self.Lmax)
            yield x[:, :-1], x[:, 1:]

def train_repeat(model, num_steps, Lmin, Lmax, eval_every=50):
    """Train with Adam; return (first, later) copy losses per round."""
    data, device = RepeatedTokens(Lmin, Lmax), d2l.try_gpu()
    optimizer = torch.optim.Adam(model.parameters(), lr=1e-3)
    history = []
    for _ in range(num_steps // eval_every):
        d2l.train_steps(model, data, optimizer, eval_every)
        x, L = repeated_batch(256, 64, 64, Lmin, Lmax, device)
        history.append(eval_copy(model, x, L)[:2])
    return history
```

10.6.2.2 A Fixed-Offset Solution

We start with the simplest version of the task: every pattern has length 32, so each sequence is one pattern followed by a copy of it. The analysis above rules out a one-layer solution that matches content. We train a one-block model on this task anyway and test it. For this experiment alone, the projections have biases (`bias=True`); their role becomes clear below.

```
model_shortcut = d2l.TinyCharLM(vocab_size=64, num_blks=1, pos='rope',
                                bias=True)
train_repeat(model_shortcut, 800, Lmin=32, Lmax=32)
first, rep, acc = eval_copy(model_shortcut, *repeated_batch(
    256, 64, 64, 32, 32, d2l.try_gpu()))
print(f'first copy {first:.2f} nats, second copy {rep:.2f} nats, '
      f'second-copy accuracy {acc:.2f}')
```

```
first copy 4.41 nats, second copy 0.09 nats, second-copy accuracy 1.00
```

The one-block model predicts the second copy almost perfectly, but it cannot be running the content-based circuit. When the period is always 32, the source of every prediction in the second copy lies 31 positions back, and a RoPE head can attend to a fixed relative offset without comparing any content (Section 10.4). The following probe tests this. It evaluates the same model on patterns of length 24, where the source lies 23 positions back, and mea-

sures each head's average attention weight at offsets 31 and 23.

```
x, L = repeated_batch(256, 64, 64, 24, 24, d2l.try_gpu())
first, rep, acc = eval_copy(model_shortcut, x, L)
with torch.inference_mode():
    m = model_shortcut.attention_weights(x[:, :-1])[0]
print(f'patterns of length 24: second-copy accuracy {acc:.2f}')
for offset in (31, 23):
    mass = m.diagonal(-offset, dim1=2, dim2=3).mean((0, 2))
    print(f'mass at offset {offset}, per head: '
          + ' '.join(f'{v:.2f}' for v in mass.tolist()))
```

```
patterns of length 24: second-copy accuracy 0.02
mass at offset 31, per head: 1.00 0.90 1.00 0.96
mass at offset 23, per head: 0.00 0.01 0.00 0.00
```

Accuracy falls to about chance, $1/64$, because the heads keep their attention at offset 31 and almost none of it moves to offset 23, where the new period puts the source. The model has learned a fixed-offset rule, not induction. The `qkv` bias makes this rule easy to express. It adds the same vector to every query and to every key, and the RoPE score of a constant query–key pair depends only on the offset. A single bias pair can therefore place a peak at a fixed distance whatever the tokens are (Exercise 4 removes the bias). Every model analyzed below has no biases, so the circuit algebra above holds for it without correction.

The general point is that a synthetic benchmark can admit a simpler solution than the mechanism it was designed to test. We therefore draw the pattern length uniformly from 16 to 32. No fixed offset then works, and predicting the later copies requires matching content against an earlier occurrence.

10.6.2.3 A Two-Block Induction Circuit

With variable pattern lengths, we train two models for 5,000 steps each: a two-block model, the smallest that can express induction, and a one-block control. Both are then evaluated on the same fresh batch.

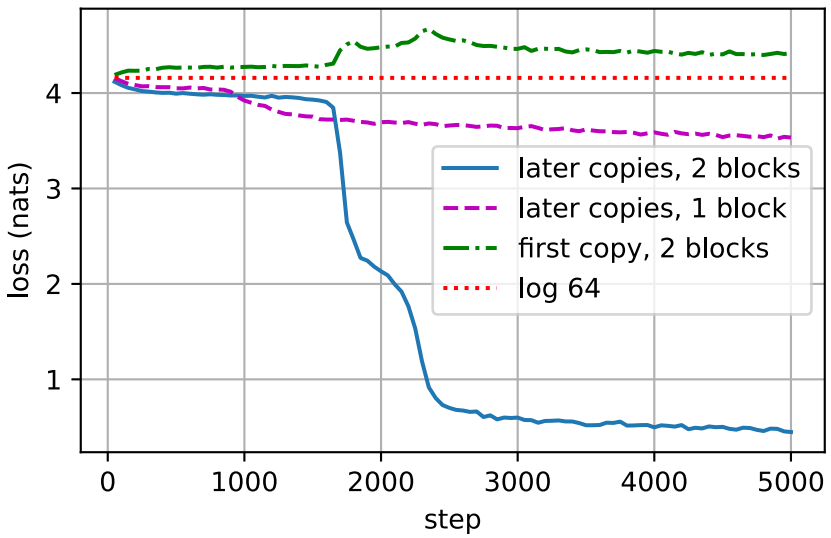
```
model_two = d2l.TinyCharLM(vocab_size=64, num_blks=2, pos='rope')
hist_two = train_repeat(model_two, 5000, Lmin=16, Lmax=32)
model_one = d2l.TinyCharLM(vocab_size=64, num_blks=1, pos='rope')
hist_one = train_repeat(model_one, 5000, Lmin=16, Lmax=32)
x, L = repeated_batch(256, 64, 64, 16, 32, d2l.try_gpu())
for name, model in (('2 blocks', model_two), ('1 block', model_one)):
    first, rep, acc = eval_copy(model, x, L)
    print(f'{name}: first copy {first:.2f}, later copies {rep:.2f}, '
          f'accuracy {acc:.2f}')
```

```
2 blocks: first copy 4.42, later copies 0.44, accuracy 0.91
1 block: first copy 4.32, later copies 3.53, accuracy 0.12
```

The results agree with the composition argument. The two-block model brings the later-copy loss down to a few tenths of a nat in a typical run and predicts about nine in ten repeated tokens; an occasional run stalls near one nat. The one-block model stays above three nats and predicts fewer than one in five. That is better than chance, because copying heads

can concentrate probability on the tokens present in the context, but it is far from retrieval. On the first copy, both models stay at or somewhat above the chance loss of $\log 64 \approx 4.16$ nats. A model that expects repetition assigns some probability to the context's tokens, and that probability is wasted whenever a newly sampled token does not repeat. The training curves show when the gap opens.

```
steps = list(range(50, 5001, 50))
d2l.plot(steps, [[h[1] for h in hist_two], [h[1] for h in hist_one],
                 [h[0] for h in hist_two], [math.log(64)] * len(steps)],
         'step', 'loss (nats)',
         legend=['later copies, 2 blocks', 'later copies, 1 block',
                 'first copy, 2 blocks', 'log 64'], figsize=(5, 3))
```



For a long stretch of training, the later-copy curve of the two-block model stays at or slightly above that of the one-block model. It then drops by several nats, most of the way within a few hundred steps and sometimes in two stages, as the computation the model performs changes abruptly. The point at which the drop happens varies from run to run by up to a thousand steps. Larger language models show a similar abrupt change in their training loss, and it coincides with the formation of induction heads (Olsson et al. 2022).

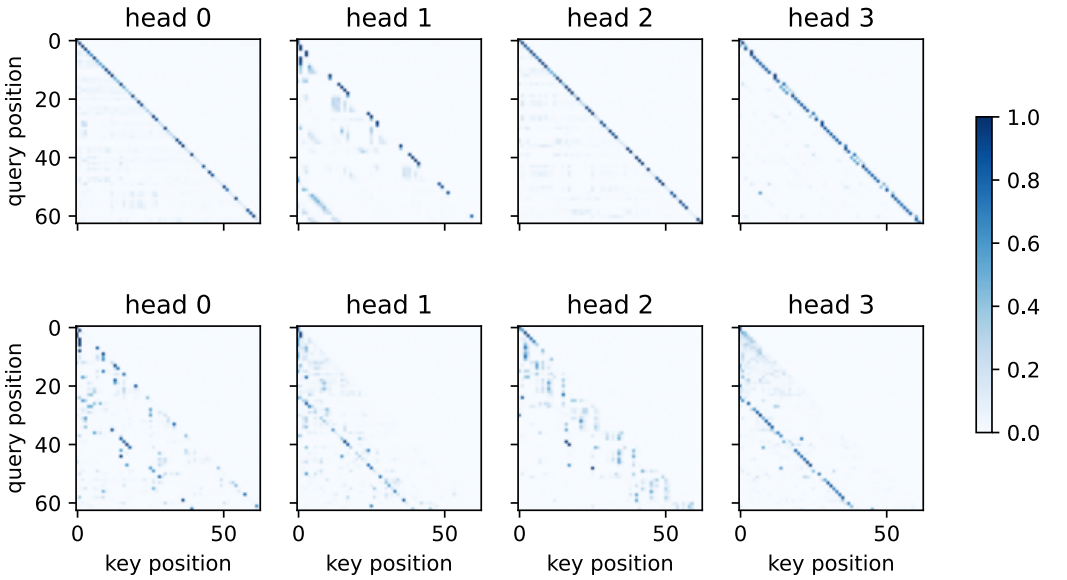
10.6.3 Evidence for the Circuit

The loss shows that two blocks solve the task; it does not show how. Three tests look inside the trained model. The attention maps show where each head looks, the OV circuits show what the heads move, and removing heads shows which of them the predictions need.

10.6.3.1 Attention Patterns of the Learned Heads

`TinyCharLM.attention_weights` returns the attention map of every head in every block. We feed one sequence with pattern length 24 and display all eight heads.

```
x, L = repeated_batch(1, 64, 64, 24, 24, d2l.try_gpu())
with torch.inference_mode():
    maps = torch.stack(model_two.attention_weights(x[:, :-1]))
d2l.show_heatmaps(maps[:, 0], xlabel='key position',
                  ylabel='query position',
                  titles=[f'head {h}' for h in range(4)],
                  figsize=(9, 4.5), cmap='Blues')
```



The top row shows block 1 and the bottom row block 2, and both halves of the predicted circuit are visible. In block 1, at least one head draws a line one step below the diagonal: a previous-token head. The others draw mostly the diagonal itself, attending to their own position. In block 2, heads draw a line 23 steps below the diagonal, starting where the second copy starts. Each query there attends to the token *after* the previous occurrence of the query's own token, which for a period of 24 lies 23 positions back. This is the induction stripe. To measure it, `head_scores` computes, for every head, the average attention on the previous token and on the induction target $j = i + 1 - L$, over a batch with pattern lengths from 16 to 32. The target score averages over the queries whose token has occurred before, $i \geq L$.

```
@torch.inference_mode()
def head_scores(model, x, L):
    """Mean attention on the previous token and on the induction target:
    shape (blocks, 2, heads)."""
    maps = torch.stack(model.attention_weights(x[:, :-1]))
    i = torch.arange(maps.shape[-1], device=x.device)
    tgt = i + 1 - L[:, None]              # Induction target of query i
    is_tgt = i == tgt[..., None]          # (batch, query, key)
    valid = tgt >= 1                      # Query token occurred before
    prev = maps.diagonal(-1, dim1=3, dim2=4).mean((1, 3))
    ind = (maps * is_tgt[:, None]).sum(-1)  # (blocks, batch, heads, query)
```

```
        ind = (ind * valid[:, None]).sum((1, 3)) / valid.sum()
        return torch.stack((prev, ind), 1)

    scores = head_scores(model_two, *repeated_batch(256, 64, 64, 16, 32,
                                                    d2l.try_gpu()))
    for b, rows in enumerate(scores.tolist()):
        for name, row in zip(('previous', 'induction'), rows):
            print(f'block {b + 1}, {name:<9}: '
                  + ' '.join(f'{v:.2f}' for v in row))
```

```
block 1, previous : 0.02 0.49 0.01 0.45
block 1, induction: 0.01 0.01 0.01 0.00
block 2, previous : 0.05 0.05 0.06 0.06
block 2, induction: 0.39 0.39 0.23 0.33
```

Each line lists the four heads of one block. In block 1, one head, or in many runs two or three that split the role, puts much of its attention on the previous token and almost none on the induction target. The other heads of the block do neither. Block 2's heads do the reverse. They share the induction pattern, and the strongest puts more than a quarter of its attention, in many runs over half, on the single position the algorithm calls for, out of up to 63 candidates. A uniform weighting would give that position about one fortieth.

10.6.3.2 The Copy Matrix

The attention maps show where the heads look; the OV circuits show what they move. If block 2's heads copy, attending to token a should raise the logit of token a itself. Write

$$C_{ab} = \mathbf{e}_b^\top \mathbf{W}_{\mathrm{OV}} \bar{\mathbf{h}}_a \tag{10.37}$$

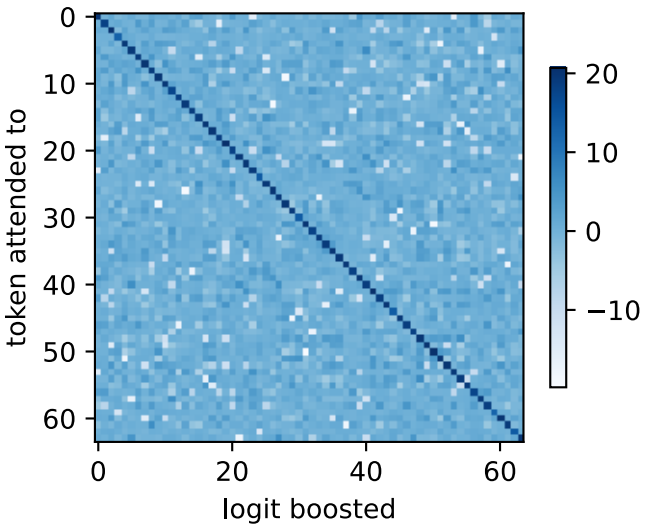
for the boost that attending to token a gives to the logit of token b. Here $\bar{\mathbf{h}}_a$ is what the head reads at a position that holds token a: the stream vector entering the block, averaged over the positions of a in a batch. In the first block it is the embedding $\mathbf{e}_a$; in block 2 it also holds what block 1 wrote. Copying means that every row of the 64×64 matrix $\mathbf{C}$ is largest on its diagonal. Because block 2's heads share the copying, we sum their OV circuits and count the rows whose largest entry is the diagonal. The same count for the one-block model serves as a control, since that model cannot retrieve.

```
@torch.inference_mode()
def copy_matrix(model, blk, X):
    """C[a, b] = e_b^T W_OV h_a: W_OV summed over the heads of blk, h_a the
    mean stream vector entering blk at the positions of token a in X."""
    E = model.token_emb.weight
    S = model.stream(X, blk)[0]                # The stream entering blk
    h = torch.zeros_like(E).index_add_(0, X.flatten(), S.flatten(0, 1))
    h = h / torch.bincount(X.flatten(), minlength=len(E))[:, None]
    W_OV = circuits(model, blk)[1].sum(0)
    return h @ W_OV.T @ E.T

x, L = repeated_batch(256, 64, 64, 16, 32, d2l.try_gpu())
for name, model, blk in (('2 blocks, block 2', model_two, 1),
```

```
                        ('1 block', model_one, 0)):
    C = copy_matrix(model, blk, x)
    frac = (C.argmax(-1) == torch.arange(len(C), device=C.device)).float()
    print(f'{name}: rows whose largest entry is the diagonal '
          f'{frac.mean():.2f} (chance 1/64)')
d2l.show_heatmaps(copy_matrix(model_two, 1, x)[None, None].cpu(),
                  xlabel='logit boosted', ylabel='token attended to',
                  figsize=(3.6, 3.6), cmap='Blues')
```

```
2 blocks, block 2: rows whose largest entry is the diagonal 1.00 (chance 1/64)
1 block: rows whose largest entry is the diagonal 0.91 (chance 1/64)
```



In block 2 of the two-block model, the largest entry of most rows lies on the diagonal, whereas a random argmax would hit the diagonal in one row out of 64. Attending to a token raises that token's own logit more than any other. The heatmap shows the same thing as a bright diagonal. The average over the stream matters. With the raw embeddings $\mathbf{e}_a$ in place of $\bar{\mathbf{h}}_a$, the same count for block 2 varies widely from run to run, because block 1 adds to each position a transformed copy of its own token, and block 2 has learned to read that copy. The control copies too. The one-block model's heads also raise the logit of the token they attend to, which is why its later-copy loss falls below chance, yet the model cannot retrieve. Copying OV circuits alone therefore do not explain retrieval. What the two-block model adds is the match on the QK side, which tells its copying heads where to look (Exercise 9 composes block 1's OV circuit into block 2's QK circuit). Neither the attention pattern nor the copy matrix shows that the model needs these heads. That is a separate question.

10.6.3.3 Head Ablation

A head with the right pattern and the right OV circuit may still be one the model does not rely on. The direct test removes the head and measures what breaks. Since `proj` is the only path from a head into the stream, zeroing the head's columns of `proj` deletes its term from Equation 10.34 and leaves every other head in that block unchanged. The function `ablate` returns a copy of the model with the given heads of one block removed.

```
def ablate(model, blk, heads):
    """Copy of model with the given heads of block blk removed."""
    model = copy.deepcopy(model)
    W_o = model.blks[blk]['proj'].weight   # (D, D): heads side by side
    with torch.no_grad():
        W_o.view(W_o.shape[0], model.num_heads, -1)[:, heads] = 0
    return model
```

The head scores pick the candidates. In block 1 we count every head whose previous-token score exceeds 0.2 as a previous-token head, which catches every head that shares the role; the threshold falls in the gap between their scores and those of the heads that attend to their own position. We remove the previous-token heads together and, as a control, all the other heads of block 1 together. In block 2 we remove the heads cumulatively, in order of their induction scores, one, two, three, and all four. The one-block model is listed for reference, and every variant is evaluated on the same fresh batch.

```
H = model_two.num_heads
prev = [h for h in range(H) if scores[0, 0, h] > 0.2]  # Previous-token heads
rest = [h for h in range(H) if h not in prev]
order = scores[1, 1].argsort(descending=True).tolist()  # Strongest first
variants = {'none': model_two,
            'block 1, previous-token heads': ablate(model_two, 0, prev),
            'block 1, all other heads': ablate(model_two, 0, rest)}
for k in range(1, H + 1):
    variants[f'block 2, top {k} induction'] = ablate(model_two, 1, order[:k])
variants['one-block model'] = model_one
x, L = repeated_batch(256, 64, 64, 16, 32, d2l.try_gpu())
print(f'block 1, previous-token heads: {prev}, others: {rest}')
print(f'{"heads removed":<31}later-copy loss  accuracy')
for name, model in variants.items():
    first, rep, acc = eval_copy(model, x, L)
    print(f'{name:<31}{rep:15.2f}{acc:10.2f}')
```

```
block 1, previous-token heads: [1, 3], others: [0, 2]
heads removed                  later-copy loss  accuracy
none                                      0.44      0.90
block 1, previous-token heads             4.73      0.03
block 1, all other heads                  3.53      0.22
block 2, top 1 induction                  0.70      0.86
block 2, top 2 induction                  1.88      0.66
block 2, top 3 induction                  3.40      0.27
block 2, top 4 induction                  4.30      0.01
one-block model                           3.56      0.11
```

Removing block 1's previous-token heads breaks retrieval. The later-copy loss rises by several nats and the accuracy falls far below that of the intact model, because block 2 has nothing left to match against. Removing the other heads of block 1 breaks it about as badly, although their maps looked irrelevant. They attend to their own position and write a transformed copy of the current token into the stream, which block 2 reads, as the copy matrix above showed. The maps had no way to show this; only the intervention did. Block 2's heads share the work. Removing the strongest alone costs only part of the accuracy, removing a second and a third costs more, and with all four removed the later-copy loss lies above even

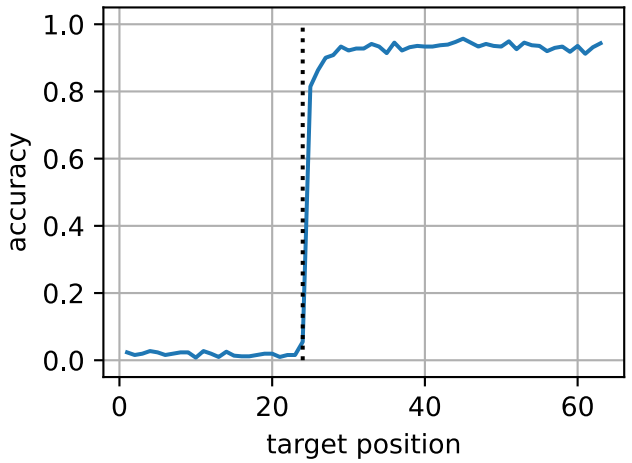
that of the one-block model. Each induction head can partly stand in for the others, so removing any single one underestimates how much the group matters. The ablations give the causal evidence that the maps could not give. To predict later copies, the model needs all of block 1, the previous-token heads and the heads that copy the current token, and the induction heads of block 2.

10.6.4 In-Context Learning

10.6.4.1 Generalization to Unseen Patterns

Every test sequence is sampled at evaluation time, so the pattern it completes has almost surely never occurred in training, since there are at least 64^{16} patterns. Across training sequences, moreover, every pair of adjacent tokens is equally frequent. The parameters therefore cannot store which token follows which in this pattern. The match-and-copy circuit instead binds each token to its successor at inference time, inside the context window. This is *in-context learning* in miniature. The model learns each new pattern from a single exposure, without a gradient step. Accuracy per position on patterns of period 24 shows the single exposure directly.

```
x, L = repeated_batch(512, 64, 64, 24, 24, d2l.try_gpu())
with torch.inference_mode():
    acc = (model_two(x[:, :-1]).argmax(-1) == x[:, 1:]).float().mean(0)
d2l.plot([torch.arange(1, 64), [24, 24]], [acc.cpu(), [0, 1]],
         'target position', 'accuracy', fmts=('-', 'k:'))
```



Accuracy is near zero across the first copy and stays low at position 24 itself, where the dotted line marks the restart. Nothing in the context yet announces that repetition has begun, so no predictor can be confident of the restart there. The low accuracy is still a shortfall of `model_two` rather than a limit set by the context (Exercise 7 computes the limit). From the next position on, accuracy is high, about 0.9 in a typical run, and it stays there for the rest of the sequence. One exposure to a pattern suffices to complete its second occurrence. The remaining errors have a cause that follows from the circuit. When a token occurs twice inside one pattern with different successors, a match on a single token is ambiguous, and our two-block circuit matches on exactly one preceding token. Disambiguating would re-

quire matching a longer prefix, through more block-1 heads at fixed offsets or through more layers.

10.6.4.2 Induction Heads in Larger Models

The same circuit appears far beyond this 64-token model. Olsson et al. (2022) found induction heads in transformer language models of many sizes, using the probe of this section: score each head by its attention from the current token to the token after that token's previous occurrence, on repeated random sequences. One of their observations matches what we measured: induction heads form abruptly, and their formation coincides with a visible change in the loss curve. Two others go beyond our experiment. Models gain most of their ability to use context in the same window, measured as the drop in loss from early to late positions of the context, and removing induction heads after training removes much of that gain in small models (Exercise 10 tests this on text). And heads found on repeated random tokens also complete structured patterns and copy fragments of text, so prefix matching is a general algorithm, not a trick for exact repetition. In models with feed-forward layers and normalization, the attribution of behavior to single components is less direct, but the circuit found in `TinyCharLM` remains a small, explicit model of how earlier context shapes later predictions.

10.6.4.3 In-Context Regression

Induction heads learn from the context by copying. Large models also learn from the context when nothing is copied, for instance by inferring a rule from a few labeled examples. The simplest such task is linear regression in context. The sequence holds pairs $(\mathbf{x}_i, y_i)$ with $y_i = \mathbf{w}^\top \mathbf{x}_i$ for a weight vector $\mathbf{w}$ drawn afresh for every sequence, followed by a query $\mathbf{x}_q$ whose label the model must predict. One step of gradient descent on the squared loss $\frac{1}{2}\sum_{i=1}^{n}(y_i - \mathbf{w}^\top \mathbf{x}_i)^2$, started at $\mathbf{w} = \mathbf{0}$ with learning rate η, gives $\mathbf{w}_1 = \eta \sum_i y_i \mathbf{x}_i$ and the prediction

$$\hat{y}_q = \mathbf{w}_1^\top \mathbf{x}_q = \eta \sum_{i=1}^{n} y_i \, \mathbf{x}_i^\top \mathbf{x}_q. \tag{10.38}$$

The right-hand side has the form of attention without a softmax: the query $\mathbf{x}_q$ is compared with the keys $\mathbf{x}_i$ by inner products, and the scores weight the values y_i. It is the numerator of the linear attention Equation 10.30 of Section 10.5 with $\phi(\mathbf{x}) = \mathbf{x}$ and no normalizer. A single head computes it from a sequence of tokens. Give each example the token $\mathbf{z}_i = [\mathbf{x}_i; y_i] \in \mathbb{R}^{p+1}$ and the query the token $\mathbf{z}_q = [\mathbf{x}_q; 0]$. Let the head's query and key maps both be $[\mathbf{I}_p \; \mathbf{0}]$, which read the input part of a token, let its value map $[\mathbf{0}^\top \; 1]$ read the label, and drop the softmax. Its output at the query token is then $\eta \sum_i (\mathbf{x}_q^\top \mathbf{x}_i)\, y_i$, which is Equation 10.38. The query token also attends to itself, but its value is zero. The function `gd_head` builds the tokens and runs this head, and the cell compares its output with the prediction of one gradient step.

```
def gd_head(X, y, x_q, eta):
    """Softmax-free head on tokens [x_i; y_i] and the query token [x_q; 0]."""
    p = X.shape[-1]
    z_q = torch.cat((x_q, torch.zeros_like(x_q[..., :1])), -1)
```

```
    Z = torch.cat((torch.cat((X, y[..., None]), -1), z_q[..., None, :]), -2)
    W_Q = W_K = torch.eye(p, p + 1)        # Queries and keys read x
    W_V = torch.eye(p + 1)[p:]             # Values read y
    Q, K, V = Z @ W_Q.T, Z @ W_K.T, Z @ W_V.T
    return eta * (Q[..., -1:, :] @ K.mT @ V)[..., 0, 0]  # At the query

n, p, eta = 32, 8, 0.1
X, w, x_q = torch.randn(n, p), torch.randn(p), torch.randn(p)
y = X @ w                                  # Context labels
w_1 = eta * X.T @ y                        # One gradient step from w = 0
print(f'one gradient step: {x_q @ w_1:.4f}, '
      f'attention head: {gd_head(X, y, x_q, eta):.4f}')
```

```
one gradient step: -0.7168, attention head: -0.7168
```

The two predictions agree to rounding error. The head's weights contain no information about $\mathbf{w}$. The rule comes from the examples in its context. With $\eta = 1/n$ the step gives $\mathbf{w}_1 = \frac{1}{n}\sum_i \mathbf{x}_i\mathbf{x}_i^\top\mathbf{w}$, and for standard normal inputs the average $\frac{1}{n}\sum_i \mathbf{x}_i\mathbf{x}_i^\top$ tends to the identity as n grows. A longer context should therefore give a better prediction, with the same weights. The cell measures the error relative to the variance of the label, averaged over 500 sequences.

```
for n in (10, 100, 1000):
    X = torch.randn(500, n, p)
    w, x_q = torch.randn(500, p), torch.randn(500, p)
    y, y_q = (X @ w[..., None])[..., 0], (x_q * w).sum(-1)
    err = ((gd_head(X, y, x_q, 1 / n) - y_q) ** 2).mean() / (y_q ** 2).mean()
    print(f'{n:4d} examples in context: relative squared error {err:.3f}')
```

```
  10 examples in context: relative squared error 0.802
 100 examples in context: relative squared error 0.096
1000 examples in context: relative squared error 0.008
```

The error falls about tenfold for every tenfold increase of n. The head learns the regression from its context, and more examples teach it more. Oswald et al. (2023) built such layers by hand and found that linear-attention transformers trained on in-context regression learn this solution, with deeper models learning refinements of it. Whether pretrained language models learn in context the same way is disputed (Shen et al. 2024). The induction head remains the best-established mechanism.

10.6.5 Attention Weights as Explanations

An attention map fixes the mixture α_{ij}, but the head's effect runs through its OV circuit. A head that attends sharply to a token might move the token's identity, some feature of it, or, if $\mathbf{W}_{\mathrm{OV}}$ annihilates the relevant directions, nothing at all. Conversely, diffuse maps can implement a precise computation. In Section 10.3, two heads blend the same two values with different weights, neither of them sharp, and a linear readout of the two mixtures recovers both values exactly.

Trained language models offer a common example of a sharp map with little effect. Many of their heads put most of their attention on the first token of the sequence, whatever that token is (Xiao et al. 2024). The softmax forces every row of weights to sum to one, so a

head with nothing to retrieve must put its weight somewhere, and the first token, visible to every query, becomes the place. This *attention sink* has a value vector of small norm, so the weight placed on it adds almost nothing to the stream (Gu et al. 2025). Read alone, the map says the first token matters most. Read with the OV circuit, it says the head is mostly idle. Section 11.3 shows why a model's sinks must be kept when its memory of the context is truncated.

The same mismatch between weights and effect appears in trained classifiers. Jain and Wallace (2019) showed that, on many text-classification tasks, quite different attention distributions leave a model's predictions unchanged. Wiegreffe and Pinter (2019) replied that adversarially chosen alternatives do not refute every explanatory use of attention, and the debate sharpened rather than settled. As an unqualified claim, "the attention weights explain the prediction" is false. A useful interpretation includes the value transformation that follows the weights, and a test of whether the model uses the computation at all.

The evidence for the induction circuit in this section comes in levels of increasing strength, summarized in Table 10.2. The lower loss of the deeper model is *behavioral* evidence that depth enables the task, but it identifies no component. Changing the period is an intervention on the input, which separated content matching from a positional shortcut. The attention pattern and the copy matrix are evidence from activations and weights. They show that candidate heads have the right form, not that the prediction depends on them. Ablation intervenes on the components themselves and shows that it does.

Table 10.2: Evidence for the induction-head interpretation, by level.

check in this section	evidence level	supported conclusion	remaining limitation
later-copy loss	behavioral	two blocks solve the repeated-token task, one does not	identifies no component
period change	input intervention	the fixed-period model uses position, not content	changes the task as well as the input
head scores	activation	heads attend with the predicted offsets	omits the value transformation
copy matrix	weights and mean activations	block 2's OV circuits copy tokens	also present in a model that does not retrieve
head ablation	component intervention	later-copy prediction needs all of block 1, not only its previous-token heads, and block 2's induction heads	zeroing moves the stream off its training distribution

Ablation asks whether a component is necessary. *Activation patching* asks what a component carries. It copies the component's output from a run on one input into a run on another and measures how much of the first run's behavior follows (K. Meng et al. 2022). With such interventions, Wang et al. (2023) traced the circuit by which GPT-2 small finds the indirect object of a sentence, a circuit that includes induction heads.

In an attention-only model these tests are comparatively direct, because fixing the attention patterns makes the stream update linear. The next chapter adds feed-forward layers and normalization to every block (Chapter 11), which removes that simplification. Large models also represent more features than the stream has dimensions, so features share directions (*superposition*) and a head rarely reads one clean feature. Analyzing them takes the same three kinds of evidence, from behavior, from weights, and from interventions, at a much larger scale.

10.6.6 Summary

An attention head consists of two circuits: a QK circuit $\mathbf{W}_q^\top \mathbf{W}_k$ that decides where it attends and an OV circuit $\mathbf{W}_o \mathbf{W}_v$ that decides what it adds to the residual stream, both of rank at most d_h and both readable from the weights. In an attention-only transformer, expanding the stream shows that one layer adds skip-trigrams such as copying to the bigram statistics of the embeddings, while predicting what followed the previous occurrence of the current token takes two layers: a previous-token head composed with a match-and-copy induction head. Trained on repeated random tokens with a variable period, a two-block `TinyCharLM` learns this circuit after an abrupt drop in loss and completes patterns it has never seen, a small instance of in-context learning. With a fixed period, a one-block model finds a positional shortcut instead. Attention maps located the candidate heads, the copy matrix showed that they copy, as even the one-block model's heads do, and ablation showed that the model needs them, together with block-1 heads whose maps looked irrelevant. Attention weights alone explain little until the value transformation and an intervention are added.

10.6.7 Exercises

1. **Tied bigrams.** With zero blocks, the logit of token b after token a is $\mathbf{e}_b^\top \mathbf{e}_a$, so the geometry of the embeddings alone fixes which successor the model prefers. Call b the *top successor* of a when $\mathbf{e}_b^\top \mathbf{e}_a$ is the largest logit of that row, and assume that there are no ties.
   1. Show that if all embeddings have the same norm, the top successor of every token is the token itself.
   2. Drop the equal-norm assumption. Show that a token $b \neq a$ can be the top successor of a only if $\|\mathbf{e}_b\| > \|\mathbf{e}_a\|$, and conclude that the top successors cannot form a cycle $a_1 \to a_2 \to \cdots \to a_k \to a_1$ with $k \geq 2$.
   3. Suppose that in a corpus the most frequent successor of the character `t` is `h`, that of `h` is `e`, that of `e` is the space, and that of the space is `t`. Show that no zero-block model reproduces all four as top successors, however long it is trained.
   4. In a one-block model, every head at the first position attends only to that position. Write the logits there as $\mathbf{EME}_{x_0}$ for a matrix $\mathbf{M}$ built from the OV circuits, and explain why neither constraint of the previous parts applies to $\mathbf{EME}^\top$.
2. **Paths through two blocks.** Fix the attention patterns of a model with L blocks of H heads on a sequence of T tokens, and write $\mathbf{A}^{\ell m}$ for the $T \times T$ matrix with entries $\alpha_{ij}^{\ell m}$.
   1. Expand the logits of Equation 10.34 for $L = 2$ into a sum of path terms. Show that the path through head m of block 1 and head m' of block 2 acts like a single head with pattern $\mathbf{A}^{2m'}\mathbf{A}^{1m}$ and OV circuit $\mathbf{W}_{\mathrm{OV}}^{2m'}\mathbf{W}_{\mathrm{OV}}^{1m}$, and bound the rank of that circuit.
   2. Count the path terms for L blocks, and evaluate the count for `model_two`.

   3. In a trained model the patterns of block 2 are not fixed: they depend on $\mathbf{h}^{(1)}$ through the QK circuit of block 2. Substitute $\mathbf{h}_i^{(1)}$ and $\mathbf{h}_j^{(1)}$ into the score of Equation 10.35 and sort the terms by whether the query side, the key side, or both contain the output of a block-1 head. These terms are called *Q-composition* and *K-composition*, and the path of the first part is *V-composition* (Elhage et al. 2021). Which kind of term does the induction circuit of Figure 10.13 rely on?
   4. Show that two blocks can match on the two-token prefix (x_{i-1}, x_i) when block 1 contains heads that attend to offsets -1 and -2, and name the composition term that carries each half of the match.
3. code **A copying head by hand.** The skip-trigram "[A] ... [A] $\rightarrow$ [A]" of Equation 10.36 can be built without training. Instantiate `d2l.TinyCharLM(64, num_blks=1, pos='none')` and overwrite the slices of head 0 in the fused `qkv` weight and in `proj`: set both the query and the key map to $c\mathbf{Q}$ for a random $d_h \times d$ matrix $\mathbf{Q}$ and a constant c, set the value map to a second random matrix $\mathbf{V}$ and the head's output map to $\mathbf{V}^\top$, and zero the output maps of the other heads.
   1. Explain why the score $\mathbf{e}_a^\top \mathbf{W}_{\text{QK}} \mathbf{e}_b$ is large exactly when $a = b$, and why $\mathbf{W}_{\text{OV}} = \mathbf{V}^\top \mathbf{V}$ raises the logit of the attended token more than any other logit.
   2. Choose c so that the keys holding the query's token receive more than 0.9 of the weight at every position of random 63-token sequences, and verify that the head's contribution to the logits is largest at the current token.
   3. Predict the later-copy accuracy of this head on the repeated-sequence task, then measure it with `eval_copy`.
   4. With $c = 0$ the head weights all positions $j \leq i$ equally, so it raises the logit of each context token roughly in proportion to the token's count. As a reference for this behavior, compute the later-copy loss and accuracy of a predictor proportional to the token counts of the context, pooled over pattern lengths uniform in 16 to 32 as `eval_copy` pools them. Compare both numbers with `eval_copy` of `model_one`, and name the information that the trained one-block RoPE model can use and this head cannot.
4. code **The shortcut without biases.** The fixed-offset model of Section 10.6.2.2 has projection biases.
   1. Let a head's query and key before rotation be the same constant vectors $\mathbf{q}$ and $\mathbf{k}$ at every position. Show with Equation 10.23 that the score depends only on the offset $j - i$, and explain how $\mathbf{q}$ and $\mathbf{k}$ can be chosen so that the score is largest at offset -31.
   2. Without biases, every query and key is a linear function of the stream. What must the token embeddings contain for a head to produce nearly constant queries and keys?
   3. Train the one-block model of the text with `bias=False` three times and report the second-copy accuracy and each head's attention at offset 31. How reliably does the shortcut form without biases, and does it survive the period-24 probe?
5. **Weights that change nothing.** The output of a head at query i is $\mathbf{W}_{\text{OV}} \sum_j \alpha_{ij} \mathbf{h}_j$ by Equation 10.34, and the softmax makes every weight α_{ij} on an unmasked key positive.
   1. Show that the attention distributions over n keys that give the same head output as α_i contain, around α_i, a piece of an affine subspace of dimension at least $n - 1 - d_h$. Evaluate the bound for the last query of a 63-token input to `model_two`.
   2. The alternatives of the first part are constructed for one input at a time, as are those of Jain and Wallace (2019) . State the condition under which a family of alternatives can be the attention patterns of one head with other QK weights on

every input at once. Explain why this condition, raised by Wiegreffe and Pinter (2019), weakens the argument that the existence of alternative distributions disqualifies attention weights as explanations.

6. code **Zero and mean ablation.** The ablations of Section 10.6.3.3 set a head's output to zero.
   1. Explain why zeroing a head's columns of `proj` removes exactly that head's contribution to the residual stream. When a block-1 head is removed, which attention patterns stay unchanged and which can change?
   2. A head's output is never zero in training, so zeroing moves the stream to states the model has not seen. *Mean ablation* replaces the output $\mathbf{o}_m$ of head m by its average $\bar{\mathbf{o}}_m$ over positions and sequences instead. Implement it: build a `TinyCharLM` with `bias=True`, copy the weights of `model_two` into it with zero `qkv` biases, zero head m's columns of `proj`, and set the bias of `proj` to the product of these columns, before zeroing, with $\bar{\mathbf{o}}_m$. Repeat the ablations of the text. Do the conclusions change?
   3. Remove the second-strongest induction head of block 2 on its own. Compare the sum of the two single-head increases in later-copy loss with the increase when both heads are removed together. What does the comparison say about how the heads share the work?
   4. Which limitations in Table 10.2 does mean ablation remove, and which remain?
7. code **The Bayes-optimal predictor.** ◻ In the task of Section 10.6.2.3, the pattern length L is uniform on $\{16, \dots, 32\}$ and the pattern tokens are uniform over 64 values. The Bayes-optimal prediction of x_t from $x_0, \dots, x_{t-1}$ averages the predictions of all pattern lengths L', weighted by their posterior probabilities, and no model achieves a lower expected loss.
   1. Show that the prefix $x_0, \dots, x_{t-1}$ has probability $64^{-\min(L', t)}$ under every length L' with which it is consistent. Neglecting shorter lengths that are consistent by chance, derive the posterior weight of the true length at $t = L$ and at $t = L + 1$.
   2. At $16 \leq t < L$ the true distribution of x_t is uniform. Show that the predictor's distribution there is not, and conclude that its first-copy loss exceeds $\log 64$.
   3. Simulate the predictor and report its first-copy loss, later-copy loss, and later-copy accuracy, pooled over positions as `eval_copy` pools them. Compare the results with `model_two`.
   4. Section 10.6.4.1 states that no predictor can be confident of the restart at position 24 of a period-24 pattern, and that the low accuracy of `model_two` there is a shortfall of the model. Compute the probability that the Bayes-optimal predictor assigns to the correct token there, and determine whether its most probable token is correct. Why does the match-and-copy circuit of `model_two` not predict that token?
8. code **Repeated tokens within a pattern.** ◻ Section 10.6.4.1 attributes the remaining errors of `model_two` to tokens that occur more than once in a pattern with different successors, which a match on a single token cannot tell apart.
   1. For a pattern of L tokens drawn uniformly from 64 values, compute the probability that the token at a given position occurs elsewhere in the pattern. A circuit that picks one of the K successors of the current token uniformly, when that token occurs K times in the pattern, is correct with probability $1/K$. Compute its expected accuracy $\mathbb{E}[1/K]$ in closed form, and evaluate both quantities for $L = 21$.
   2. Evaluate `model_two` on patterns of length 21, which `repeated_batch` tiles to three copies within 64 tokens. Split the later-copy targets by whether the current token

is unique in the pattern, and report the accuracy of each group on the second and on the third copy. Compare the overall accuracy with $\mathbb{E}[1/K]$.

3. Explain the results through the single-token match. What does an additional earlier occurrence with the same successor do to the attention pattern and to the output, and what does one with a different successor do?
4. Two blocks can match on the two-token prefix (x_{i-1}, x_i) when block 1 contains heads that attend to offsets -1 and -2. Measure the attention mass that each block-1 head of `model_two` places at offset -2, and decide from this and your earlier measurements whether `model_two` matches on more than one token.

9. code **K-composition in the weights.** ▯ The match of the induction head works because the QK circuit of block 2 reads what the OV circuit of block 1 wrote into the stream.
   1. For every pair of head m of block 1 and head m' of block 2 of `model_two`, compute the 64×64 matrix
   $$\mathbf{E}\,\mathbf{W}_{\mathrm{QK}}^{2m'}\,\mathbf{W}_{\mathrm{OV}}^{1m}\,\mathbf{E}^{\top},$$
   whose entry (a, b) scores a query token a against a key whose predecessor was b. Measure the fraction of rows whose largest entry lies on the diagonal, as the copy test of Equation 10.37 does for copying.
   2. Compare with the direct path $\mathbf{E}\,\mathbf{W}_{\mathrm{QK}}^{2m'}\,\mathbf{E}^{\top}$ of the same block-2 heads. Which pairs of heads carry the match, and do they agree with the attention-based scores of `head_scores`?
   3. The computation ignores the rotation that RoPE applies inside the QK circuit. Include it for a fixed offset with Equation 10.23, and check whether doing so at offset -23 changes the ranking of the pairs.

10. [extended] **Induction heads in text.** ▯▯ Olsson et al. (2022) tie the formation of induction heads in language models to an abrupt gain in in-context learning, measured as the drop in loss from early to late positions of the context. Test this on the byte-level model of Section 10.4.4.3.
    1. Train `d2l.TinyCharLM` with `pos='rope'` on the WikiText-103 bytes of Section 10.4.4.3 in 30 rounds of 100 steps of `d2l.train_steps`. After each round, record two quantities. The first is the largest induction-target score of a block-2 head on repeated random sequences over the text vocabulary with pattern lengths 16 to 32, computed with `head_scores`. The second is the in-context gain, the validation loss at target positions 0 to 15 minus that at positions 96 to 127.
    2. Plot both quantities against the training step. Do the induction score and the in-context gain rise in the same window?
    3. After training, remove the block-2 heads with the highest induction scores with `ablate`, and measure the in-context gain and the validation loss again. How much of the gain do these heads carry?
    4. The repeated random sequences contain no structure besides repetition, while text repeats words and phrases at irregular distances. Which of your results would change if the text contained no repeated words, and why?

10.7 Further Reading

The sections of this chapter teach one representative of each idea: one soft lookup, one scoring function, one way to split attention into heads, a few ways to encode position, one exact kernel and two ways to avoid the quadratic cost, and one circuit traced through a small model. This section points to topics they had no room for: other readings of the lookup and its normalization, what trained heads do and how many are needed, further

positional schemes and their limits, older and newer ways to reduce the cost, and the tools that analyze circuits in large models. Each topic states the problem it addresses and what it changes relative to the chapter. Implementations, tutorials, and courses for the material that the chapter does teach are listed in its introduction, Chapter 10. Nothing else in the book depends on this section.

10.7.1 The Lookup and Its Normalization

10.7.1.1 Attention as Associative Memory

Section 10.1 read attention as a smoothed lookup in a table of key-value pairs, with a bandwidth that decides between retrieving one entry and averaging many. Ramsauer et al. (2021) read the same computation as memory retrieval: softmax attention with one query is one update step of a modern Hopfield network whose stored patterns are the keys. The network's energy function makes the trade-off precise. At a large inverse temperature the update converges to a single stored pattern; at a small one it converges to an average of many, which is the role that the bandwidth plays in Section 10.1 and the factor $1/\sqrt{d}$ in Section 10.2. They also show that such a network can store a number of patterns exponential in the dimension.

10.7.1.2 Bandwidths That Follow the Data

The Nadaraya–Watson estimator of Section 10.1 uses one bandwidth everywhere, which is too narrow where the data are sparse and too wide where they are dense. Adaptive kernel estimators let the bandwidth follow the local density of the data (Silverman 1986), a hand-built analogue of a temperature that depends on the query. The same lookup also works as a representation without any training: ASIF (Norelli et al. 2022) describes an image or a caption by its similarities to a fixed set of image–caption anchor pairs, which lets two separately trained encoders exchange information without a joint model.

10.7.1.3 Monotone Alignment

Bahdanau's attention of Section 10.2 was not the first differentiable alignment. Graves's handwriting synthesis (Graves 2013) aligned each pen stroke with a position in the text through a soft window that could only move forward, a constraint inherited from the left-to-right hidden Markov models of speech recognition (Rabiner and Juang 1993). Dropping monotonicity is what lets attention handle reordering between languages, as in the alignment of "black cat" with "chat noir".

10.7.1.4 Weights That Need Not Sum to One

The softmax forces every row of weights to sum to one, so a query that finds nothing relevant must still put its weight somewhere. Trained models often put it on the first token, the attention sink that Section 10.3 and Section 10.6 mention and Section 11.3.4.2 measures. Several lines of work ask why. M. Sun et al. (2024) find that sinks coincide with a few activations in the residual stream that are orders of magnitude larger than the rest and act as fixed biases. Barbero, Arroyo, et al. (2025) argue that sinks are useful: they let a deep model with a long context avoid mixing every token into every other. Two designs remove the need for them. Sigmoid attention (Ramapuram et al. 2025) replaces the softmax by an elementwise sigmoid with a bias of $-\log n$, so that a query can give little weight to every key. It trains stably once that bias is in place and the attention output is kept small early in training, and

its fused kernel is about a sixth faster than FlashAttention-2 on an H100. Gated attention (Qiu et al. 2025), which Section 11.3.4.2 describes, keeps the softmax and multiplies each head's output by a sigmoid gate computed from the layer input.

10.7.2 Heads

10.7.2.1 Redundant Heads

Section 10.3 proved that one head cannot return two values and that two heads can, and the trained heads of its experiment nearly duplicated each other. Trained models at scale show the same redundancy. Michel et al. (2019) removed heads of trained translation and BERT models at test time and found that a large fraction can go with little loss, and that many layers work nearly as well with a single head. Voita et al. (2019) identified the few heads that carry the effect, among them heads that attend to the previous token, to rare words, or along a syntactic relation, and pruned most of the others with a learned gate at a small cost in quality.

10.7.2.2 Sharing Keys and Values

Section 10.3.2.5 quoted two results on grouped-query attention that seem to disagree. Ainslie et al. (2023) found eight key-value groups within a tenth of a point of full multi-head attention, and DeepSeek-V2 (DeepSeek-AI 2024) reported a clear loss for the same change. The two results come from different training. The first converted a strong multi-head checkpoint and trained it for 5% of its original budget. The second trained a 7B dense model from scratch on 1.33T tokens. Uptraining starts from heads that have already learned distinct weightings, while training from scratch must find them with fewer key-value heads from the first step. Multi-head latent attention, DeepSeek-V2's answer, is described in Section 11.8.

10.7.2.3 Subtracting Two Weightings

Differential attention (T. Ye et al. 2025) computes two softmax weightings from two sets of queries and keys and subtracts a learned multiple of the second from the first, so that weight that both place on irrelevant tokens cancels. The authors report that it matches a standard transformer with about two thirds of the parameters or training tokens. No widely used open model has adopted it so far, so it is a candidate rather than a standard.

10.7.3 Positions

10.7.3.1 Learned Relative Positions

Before RoPE, relative position entered the score as trained vectors, one per clipped offset, added to the keys inside the dot product (Shaw et al. 2018). The Music Transformer (Huang et al. 2018) made this affordable for long sequences by computing the relative logits for all offsets at once and then shifting them into place, instead of storing a vector for every pair of positions. RoPE in Section 10.4 reaches scores that depend only on the offset with no learned parameters at all.

10.7.3.2 RoPE in Trained Models

Released models differ less in whether they use RoPE than in its base. Llama 2 used 10^4, Llama 2 Long and Llama 3 use 5×10^5 (Xiong et al. 2024), and the global layers of Gemma 3 use 10^6 (Gemma Team 2025). A larger base slows every pair except the fastest, and so moves more pairs into the slow group of Section 10.4.4.4. Barbero, Vitvitskyi, et al. (2025) looked at which pairs a trained model uses. Gemma 7B uses the fastest pairs for heads that attend by position, such as previous-token heads, and the slowest pairs as nearly position-free channels for content. These two roles are the trained counterpart of the fast and slow pairs of Section 10.4. Removing the lowest frequencies altogether, a variant they call p-RoPE, helped in their experiments.

10.7.3.3 Per-Pair Rescaling

YaRN, in Section 10.4.4.4, leaves the fast pairs alone, interpolates the slow ones, and ramps between the two groups with a hand-set rule. LongRoPE (Ding et al. 2024) replaces the rule by an evolutionary search for one rescale factor per pair and leaves the first tokens unscaled. With at most a thousand fine-tuning steps at training lengths up to 256k it extends the context to 2048k tokens.

10.7.3.4 Length Generalization of ALiBi and NoPE

Section 10.4.4.3 found that ALiBi and NoPE keep their perplexity beyond the training length. T.-C. Chi et al. (2023) explain the first. ALiBi's steepest slopes make most heads attend within a window near the training length, so the extra tokens are effectively never read; the exercise on where extrapolation fails measures this. NoPE has its own limit. Haviv et al. (2022) showed that causal language models without positional encoding are competitive in perplexity and learn absolute position from the causal mask. Jie Wang et al. (2024) found that their length generalization also breaks, because attention spreads over too many tokens, and restored it with a softmax temperature per head. Llama 4 (Meta AI 2025) applies the same fix at inference time to the position-free global layers that it interleaves with RoPE layers.

10.7.3.5 Perplexity and the Use of Long Contexts

A model whose perplexity stays flat at four times its training length need not use the extra tokens. N. F. Liu et al. (2024) placed the answer to a question at different positions of a long context and found that models retrieve it far better from the beginning or the end than from the middle. How to train models that use long contexts, by continued pretraining on long documents with a raised base and by schedules that grow the length, belongs to the Language Models part of the book.

10.7.4 Cost

10.7.4.1 Approximating the Softmax Kernel

Before fused exact kernels, several methods approximated attention to reach linear cost. Performer (Choromanski et al. 2021) replaces $\exp(\mathbf{q}^\top \mathbf{k})$ by an inner product of positive random features whose expectation is the exponential, so that the associativity of Section 10.5 applies. Linformer (Wang et al. 2020) projects the keys and values along the sequence axis to a fixed length, and Reformer (Kitaev et al. 2020) hashes similar queries and keys into

shared buckets and attends within each bucket. cosFormer (Qin et al. 2022) uses ReLU features with a cosine reweighting that favors nearby positions. None of them is exact, and in practice they were displaced by exact attention in fused kernels, by windows, and by the gated recurrences below. Tay et al. (2020) catalogue dozens more.

10.7.4.2 Global Tokens and Strided Patterns

The window of Section 10.5.3.2 needs many layers before distant tokens meet. Longformer (Beltagy et al. 2020) adds to the window a few positions chosen for the task, such as a classification token, that attend to every token and are attended by every token, so that the whole document reaches them in one layer. The Sparse Transformer (Child et al. 2019) combines a local pattern with a strided one so that two layers connect any pair of positions. Both patterns are fixed in advance, like the window.

10.7.4.3 Windows in Released Models

Mistral 7B (Jiang et al. 2023) used a window of 4,096 tokens in all 32 layers, with a rolling cache that holds only the last w keys and values. Its successor, Mistral-7B-Instruct-v0.2 (Mistral AI 2023), dropped the window for a full context of 32k tokens and a RoPE base of 10^6. Gemma 2 (Gemma Team 2024) alternates windowed and global layers one to one and reports almost unchanged perplexity when its windows shrink from 4,096 to 1,024 tokens at inference. Gemma 3 (Gemma Team 2025) uses five windowed layers of 1,024 tokens per global layer, the ratio whose cache saving Section 10.5 computes, and finds little change in perplexity even at seven to one.

10.7.4.4 Sparsity Learned from Content

A window decides by position which keys a query sees. Three recent designs decide by content, and each is built to run as a fused kernel over blocks. Natively sparse attention (Yuan et al. 2025) scores compressed blocks of keys, keeps the highest-scoring blocks for each query, and adds a sliding window, all trained end to end. MoBA (E. Lu et al. 2025) routes each query to the key blocks whose mean key it matches best, a mixture-of-experts gate over blocks. DeepSeek Sparse Attention (DeepSeek-AI et al. 2025) scores all past tokens with a small separate indexer and runs attention over the top-scoring ones only.

10.7.4.5 FlashAttention-3 and FlexAttention in Detail

Section 10.5 describes FlashAttention-3 (Shah et al. 2024) as the same algorithm with a new schedule. On H100 GPUs, some warps issue asynchronous tile loads while others run the matrix products. Two groups of warps also take turns, so that the exponentials of one group, which run on a much slower unit, overlap the matrix product of the other. Its FP8 mode quantizes each block separately. It first multiplies queries and keys by a random orthogonal matrix that spreads outlying coordinates, which lowers the error of FP8 attention by a factor of 2.6 against a single scale per tensor. FlexAttention (Dong et al. 2025) generates kernels for masks and score modifications written as Python functions; on the variants that FlashAttention-2 supports, its kernels run at 0.68 to 1.43 times the speed of the handwritten one. Kernels for decoding, which split the cache across thread blocks and read it from pages, are treated with the serving systems of the Language Models part.

10.7.4.6 Gated Recurrences and Hybrids

Section 10.5 ended with the observation that plain linear attention lost to softmax attention, and that two changes made it competitive. Gated linear attention (Yang, Wang, Shen, et al. 2024) adds a decay that depends on the input, and Gated DeltaNet (S. Yang et al. 2025) replaces the additive write by a delta rule that overwrites the value stored under a key; both train with chunkwise kernels like the chunked form of the chapter. The models that use them keep a few softmax layers for recall: Qwen3-Next (Qwen Team 2025) places one gated-attention layer after every three Gated DeltaNet layers, Kimi Linear (Kimi Team 2025b) mixes three of its recurrent layers with one latent-attention layer, and MiniMax-01 (MiniMax 2025) uses one softmax layer per seven linear ones. Mamba-2 (Dao and Gu 2024) shows that a selective state space model with a scalar decay computes the same output as a masked attention whose mask holds products of the decays, so the chunked form with a decay is one of its algorithms. Chapter 12 teaches these recurrences.

10.7.5 Circuits in Large Models

10.7.5.1 Patching and Automated Circuit Discovery

The ablation of Section 10.6.3.3 shows that a component is needed, not what it carries or along which path. Activation patching copies an activation from a run on a clean input into a run on a corrupted one and measures how much of the clean behavior returns (K. Meng et al. 2022). Path patching restricts the copy to one path between components, which is how Wang et al. (2023) traced the circuit that GPT-2 small uses to complete "When Mary and John went to the store, John gave a drink to" with "Mary". ACDC (Conmy et al. 2023) automates the search over edges. Attribution patching (Nanda 2023) replaces each patch by a first-order estimate from gradients, so that one backward pass scores every component, and AtP* (Kramár et al. 2024) corrects the cases where that estimate misses an important component. Zhang and Nanda (2024) and Heimersheim and Nanda (2024) study how the conclusions depend on the choice of corrupted input and on the value that replaces an ablated component. The Language Models part of the book treats these methods on large models.

10.7.5.2 Superposition and Dictionary Features

The circuit algebra of Section 10.6 assumes that each head reads and writes identifiable directions of the residual stream. In large models the features outnumber the dimensions and share directions, a regime that Elhage et al. (2022) call superposition and reproduce in small models. Sparse dictionaries learned on the activations recover features one at a time, and attribution graphs built from them trace which features cause which across layers (Ameisen et al. 2025). These methods, too, belong to the Language Models part.

10.7.5.3 Induction Heads at Scale

Olsson et al. (2022) found the induction heads of Section 10.6 in large models as well. Heads identified on repeated random tokens also copy translated fragments and support few-shot completion, and the authors give six lines of evidence that induction heads carry most of in-context learning: causal in small attention-only models, correlational in large ones.

10.7.5.4 In-Context Learning as an Algorithm

Section 10.6 showed that one head without a softmax takes one gradient step on a regression problem stored in its context. Garg et al. (2022) trained transformers to regress in context on whole function classes, such as linear functions, sparse linear functions, and decision trees, and found errors close to those of the best algorithm for each class. Akyürek et al. (2023) asked which algorithm a trained model implements and found a progression with depth and width, from one gradient step towards ridge regression.

Chapter 11

Transformers

The language models behind chat assistants are built from the attention layers of Chapter 10. Attention alone does not make such a model: it moves information between positions, but computes no nonlinear feature of any single one. When an assistant writes a reply, each new word passes up through dozens of units of the same design, and attention is only one part of each. What else does a unit need? How can a stack that deep be trained? And how can it write fast enough for us to read along?

The unit is the *transformer block*, introduced in 2017 for machine translation (Vaswani et al. 2017). It pairs attention with a small feed-forward network that transforms each position on its own, and both add their outputs to a residual stream that carries the signal upward. With a few settings changed, it now runs language, vision and speech models.

Each setting answers a problem of depth or scale. A deep stack can lose its signal at initialization or diverge in training. Generating one token at a time repeats work unless the model keeps what it has computed, and then reading that memory, not the arithmetic, sets the speed. More parameters cost more computation unless most of them sit idle for any one token. And a run that costs millions has to be sized before it starts. We build one block in which each setting is a constructor argument, and test each on small models.

Section 11.1 asks what a block needs besides attention, and where its normalization must sit for a deep stack to train. Section 11.2 asks how a stack of blocks becomes a language model, how to train it, and whether the same code runs GPT-2. Section 11.3 asks how to generate without recomputing the context, and how to shrink the cache that generation then reads. Section 11.4 asks what changes when the causal mask is removed, or a second stack is joined by cross-attention. Section 11.5 asks whether the same blocks can classify an image cut into patches. Section 11.6 asks how a model can store many feed-forward networks and run only a few for each token. Section 11.7 asks what a training run costs, and how to split a budget between parameters and data.

The chapter treats the architecture of base models. Pretraining at scale, data, fine-tuning, serving systems, and quantization belong to the Language Models part, kernels and parallelism to Chapter 13, and recurrent alternatives to a growing cache to Chapter 12.

Resources and Further Reading

The resources below accompany the sections of this chapter; each names the section it goes with. All are freely accessible online. Related topics that the chapter does not teach are collected in Section 11.8.

Build-alongs and courses

- Let's Build GPT: From Scratch, in Code — Andrej Karpathy (2023) — the video counterpart of Section 11.2: a character-level GPT assembled and trained in real time; nanoGPT and build-nanogpt are its repository forms, and nanochat applies the same approach to a full chat system, with its training costs stated in dollars.
- The Annotated Transformer — Harvard NLP (2018, refreshed 2022) — the original encoder–decoder, implemented line by line next to the text of the paper, the wiring of Section 11.4.
- Build a Large Language Model (From Scratch) — Sebastian Raschka (2024) — code for a book-length version of Section 11.1 through Section 11.3, with notebooks for GQA, sliding windows, and from-scratch ports of current open models (the book itself is paid).
- Stanford CS336: Language Modeling from Scratch — a course whose first assignment builds RMSNorm, RoPE, SwiGLU, causal attention, and the training loop of Section 11.1 and Section 11.2 from primitives; the lectures are on YouTube.
- CMU Advanced NLP, minLlama Assignment — build a Llama-style decoder and load pretrained weights into it, the exercise of loading GPT-2 in Section 11.2 at a larger scale.

Architecture papers

- On Layer Normalization in the Transformer Architecture — Xiong et al. (2020) — the analysis of pre-norm and post-norm behind the signal-propagation experiment of Section 11.1, and the reason post-norm models need warmup, which Section 11.2 tests.
- GLU Variants Improve Transformer — Shazeer (2020) — the four-page note whose matched-parameter comparison Section 11.1 repeats in miniature.
- GQA: Training Generalized Multi-Query Transformer Models — Ainslie et al. (2023) — grouped-query attention as Section 11.3 uses it, including the uptraining recipe that converts multi-head checkpoints.
- Efficient Streaming Language Models with Attention Sinks — Xiao et al. (2024) — the first-token sink and the windowed cache that keeps it, which Section 11.3 measures on GPT-2.
- Perceiver IO: A General Architecture for Structured Inputs and Outputs — Jaegle et al. (2022) — learned queries that read an input of any size and write an output of any shape, the interface of Section 11.4.
- An Image Is Worth 16x16 Words — Dosovitskiy et al. (2021) — the ViT paper behind Section 11.5, which weighs scale against built-in assumptions; the section's small experiment asks the same question in a different regime.
- Switch Transformers — Fedus et al. (2022) and DeepSeek-V3 Technical Report — DeepSeek-AI (2024) — the two ends of the design range in Section 11.6: top-1 routing with an auxiliary balancing loss, and fine-grained experts balanced by a bias.
- The Big LLM Architecture Comparison — Sebastian Raschka (2025, maintained) — current open models compared on the same few axes, a maintained version of Table 11.2 in Section 11.7.

The arithmetic of scale

- Transformer Inference Arithmetic — kipply (2022) — per-token inference FLOPs, cache storage, and bandwidth limits, the accounting of Section 11.3, compared with a real system.
- Transformer Math 101 — EleutherAI (2023) — the training-side companion: where $6ND$ comes from and what it predicts, the count that Section 11.7 checks against the framework.
- Training Compute-Optimal Large Language Models — Hoffmann et al. (2022) — Chinchilla, the tokens-per-parameter result whose law Section 11.7 fits and whose bend its miniature study shows.
- The Ultra-Scale Playbook — Hugging Face (2025) — the parallelism and memory engineering that the budgets of Section 11.7 require beyond one GPU, treated in Chapter 13.

11.1 The Transformer Block

Chapter 10 built attention, a layer in which each position reads from all the others. On its own it makes a weak model. Each output of an attention layer is a weighted average of value vectors, which are linear projections of the inputs. The softmax weights decide how much of each value to take, but nothing in the layer computes a nonlinear feature of what a single position holds. A language model needs that computation at every position, and it needs it stacked dozens of layers deep. Depth brings a problem of its own: a signal that passes through 32 layers can shrink, explode, or lose the differences between tokens before it reaches the output, and the gradient makes the same trip in reverse.

The *transformer block* answers both needs. Attention moves information between positions, a position-wise feed-forward network transforms each position on its own, and both add their outputs to a shared residual stream that carries the signal through the stack. This section studies the two design choices that separate the 2017 block from current ones: where the normalization sits and what form the feed-forward network takes. We measure the first at initialization, where it decides whether a 32-block stack keeps its tokens distinct and its attention trainable. We measure the second in a small language model at matched parameter count. A configurable `TransformerBlock` class then collects these choices, and the rest of the chapter builds on it.

11.1.1 Components of a Transformer Block

A transformer block contains two sublayers connected by a residual stream. Section 10.6 introduced the *residual stream* view: each token carries a running vector $\mathbf{x} \in \mathbb{R}^d$ from layer to layer. Sublayers never overwrite this vector. They read from it, compute, and *add* their result back. The two sublayers of the block divide the work as follows.

- **Attention** is the only place where positions interact. Each token queries the others as in Section 10.3 and adds the retrieved mixture to its stream. Contextual information enters through this sublayer.
- The **feed-forward network** (FFN) acts on each position separately: the same two- or three-matrix MLP is applied token by token, with no interaction between positions. It transforms each token's accumulated state and contains most of the block's parameters.

Writing Attn for multi-head self-attention and Norm for a normalization layer, the modern (*pre-norm*) block computes

$$\mathbf{h} = \mathbf{x} + \mathrm{Attn}(\mathrm{Norm}_1(\mathbf{x})), \qquad \mathbf{y} = \mathbf{h} + \mathrm{FFN}(\mathrm{Norm}_2(\mathbf{h})). \tag{11.1}$$

Both sublayers preserve the shape (n, d). The block maps a sequence of d-dimensional vectors to a sequence of the same length and width, so dozens of copies can be stacked. The block also knows nothing about the order of the tokens. Attention is permutation equivariant (Section 10.4), and no other part of the block depends on order. The model around the blocks must supply positions, and the next section shows how.

Figure 11.1 shows the wiring, in the two normalization arrangements compared below.

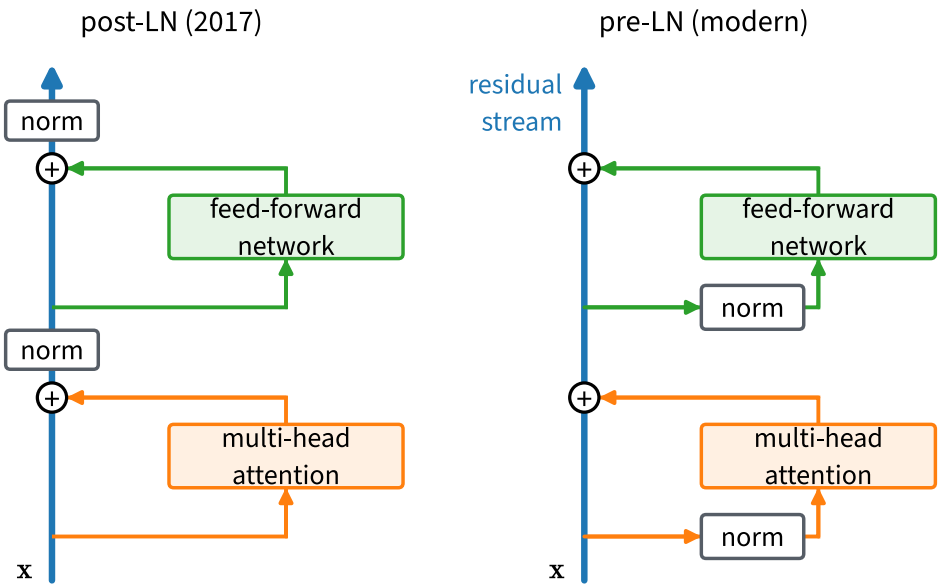


Figure 11.1: The transformer block: attention and a position-wise FFN read from the residual stream and add their results back. Left, the 2017 arrangement: normalization sits on the stream itself, after each addition. Right, the modern arrangement: normalization moves onto each branch, and the stream runs uninterrupted from input to output.

11.1.2 Normalization Placement

Residual connections and normalization entered the block together in 2017 (Vaswani et al. 2017). Without some control on scale, the sum of dozens of sublayer outputs drifts, and deep stacks become hard to train. The standard normalizer is *layer normalization* (Ba et al. 2016) (Section 7.3.4.1): it subtracts the mean of each token's d features, divides by their standard deviation, and then applies a learned scale and shift per feature. Unlike batch normalization (Section 7.3), it uses no batch statistics, so it behaves identically in training and inference and does not depend on the sequence length. The 2017 paper also fixed *where* the normalization goes, and later work revised that choice.

11.1.2.1 Two Arrangements

The original block normalizes the stream *after* each addition ("post-LN"):

$$\mathbf{h} = \mathrm{Norm}_1(\mathbf{x} + \mathrm{Attn}(\mathbf{x})), \qquad \mathbf{y} = \mathrm{Norm}_2(\mathbf{h} + \mathrm{FFN}(\mathbf{h})). \tag{11.2}$$

The modern block Equation 11.1 normalizes each branch's *input* ("pre-LN") and leaves the stream alone. In code the two differ by two lines. The minimal block below takes the arrangement as a flag. Its attention is the `d2l.MultiHeadAttention` of Section 10.3, and its FFN is the classic two-matrix MLP with the GELU activation of Section 4.1. Later in this section we upgrade both ingredients and collect every option in one configurable class; until then this short block keeps the two arrangements side by side and nothing else in view.

```
class MiniBlock(nn.Module):
    """Attention + MLP on a residual stream; norm placement as a flag."""
    def __init__(self, num_hiddens, pre_norm):
        super().__init__()
        self.pre_norm = pre_norm
        self.attention = d2l.MultiHeadAttention(num_hiddens, 4)
        self.W_1 = nn.Linear(num_hiddens, 4 * num_hiddens)
        self.W_2 = nn.Linear(4 * num_hiddens, num_hiddens)
        self.norm1 = nn.LayerNorm(num_hiddens, eps=1e-5)
        self.norm2 = nn.LayerNorm(num_hiddens, eps=1e-5)

    def ffn(self, X):  # exact GELU
        return self.W_2(F.gelu(self.W_1(X)))

    def forward(self, X):
        if self.pre_norm:
            Y = self.norm1(X)
            X = X + self.attention(Y, Y, Y)
            return X + self.ffn(self.norm2(X))
        X = self.norm1(X + self.attention(X, X, X))
        return self.norm2(X + self.ffn(X))
```

11.1.2.2 Signal Propagation at Initialization

Much of the difference between the two arrangements shows before any training. A freshly initialized network is a fixed random function, so we can *measure* how it treats signals. We stack 32 blocks, give every weight matrix the Xavier initialization of the original transformer (Section 4.4.2.1), pass a random sequence through the stack, and record three quantities at every depth: the scale of the residual stream, how *distinct* the tokens remain from one another, and how much gradient each block's attention receives from a loss at the top. The numbers change little from one random draw to the next.

For distinctness we use the cross-token spread $\|\mathbf{H} - \bar{\mathbf{H}}\| / \|\mathbf{H}\|$, where $\mathbf{H}$ holds the token vectors of one sequence as rows and $\bar{\mathbf{H}}$ copies their average to every row. The spread is about 1 for unrelated random tokens and 0 when all tokens have collapsed onto a single point. The loss takes some care. After a final normalization layer, the obvious probe $\|\mathbf{H}\|^2$ is constant by construction and has zero gradient, so we use a generic linear readout $\langle \mathbf{R}, \mathbf{H} \rangle$ for a fixed random $\mathbf{R}$ instead.

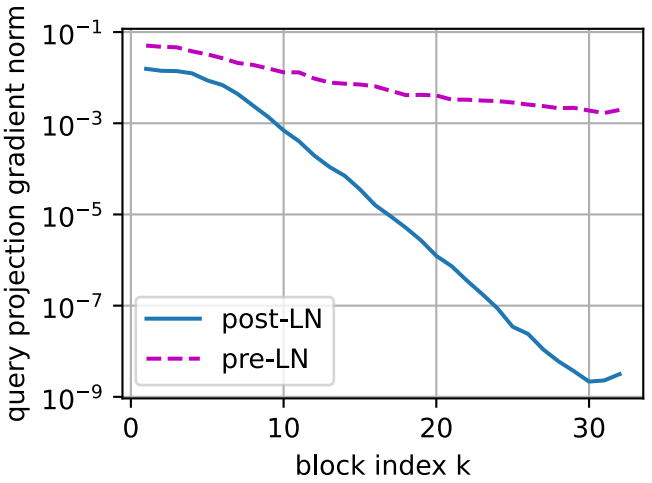
```
def signal_stats(pre_norm, num_blks=32, num_hiddens=256):
    """Token spread, stream RMS, and gradient norms of a fresh stack."""
```

```
    blks = nn.ModuleList([MiniBlock(num_hiddens, pre_norm)
                          for _ in range(num_blks)])
    for p in blks.parameters():
        if p.ndim == 2:                # the 2017 initialization
            nn.init.xavier_uniform_(p)
    norm = torch.linalg.vector_norm
    H = torch.randn(2, 64, num_hiddens)
    spread, rms = [], []
    for blk in blks:
        H = blk(H)
        spread.append(norm(H - H.mean(1, keepdim=True)) / norm(H))
        rms.append(H.pow(2).mean().sqrt())
    (H * torch.randn_like(H)).mean().backward()  # generic readout loss
    grad_q = [norm(blk.attention.W_q.weight.grad) for blk in blks]
    proj = {n: norm(getattr(blks[-1].attention, n).weight.grad).item()
            for n in ('W_q', 'W_k', 'W_v', 'W_o')}
    return {'spread': torch.stack(spread).tolist(),
            'rms': torch.stack(rms).tolist(),
            'grad_q': torch.stack(grad_q).tolist(), 'proj': proj}
```

We run it for both arrangements, print the stream RMS and the token spread at a few depths, and plot the gradient of each block's query projection:

```
stats = {pre: signal_stats(pre) for pre in (False, True)}
for pre, name in ((False, 'post-LN'), (True, 'pre-LN')):
    s = stats[pre]
    print(f'{name:>8}: stream RMS at k=32: {s["rms"][-1]:5.2f}, '
          'token spread at k=8,16,32: '
          + ' '.join(f'{s["spread"][k-1]:.1e}' for k in (8, 16, 32)))
    print('          grad norms at k=32: '
          + '  '.join(f'{n} {g:7.1e}' for n, g in s['proj'].items()))
d2l.plot(list(range(1, 33)), [stats[False]['grad_q'], stats[True]['grad_q']],
         'block index k', 'query projection gradient norm',
         legend=['post-LN', 'pre-LN'], yscale='log')
```

```
post-LN: stream RMS at k=32:  1.00, token spread at k=8,16,32: 2.9e-01 2.2e-02 1.1e-04
         grad norms at k=32: W_q 3.2e-09  W_k 2.9e-09  W_v 6.6e-02  W_o 6.1e-02
 pre-LN: stream RMS at k=32:  5.40, token spread at k=8,16,32: 7.2e-01 4.7e-01 3.5e-01
         grad norms at k=32: W_q 2.0e-03  W_k 2.0e-03  W_v 8.5e-02  W_o 8.1e-02
```



The two arrangements produce different networks from the same weights. Post-LN renormalizes the stream after every addition, so its root mean square (RMS) is pinned at 1 at every depth. Pre-LN only ever adds, so its stream grows like the square root of the depth. At initialization the block outputs are nearly uncorrelated and their variances add, which predicts an RMS after 32 blocks of $\sqrt{33} \approx 5.7$. The measured value, between 5 and 5.6, falls somewhat short; an exercise asks why. This growth stabilizes the stack. Each pre-LN branch reads the stream through its own normalization, so its output has a fixed scale. As the stream grows, the *relative* contribution of each successive block therefore shrinks.

The post-LN stack has no such damping, and the token spread shows the consequence. At initialization attention is nearly uniform, so every attention sublayer pulls each token toward the average token. Post-LN then renormalizes, which locks in the contraction, and the spread decays *geometrically*: by block 32 the tokens agree to about one part in ten thousand. Pre-LN dilutes the same contraction by the growing stream, and its spread falls slowly, to between 0.3 and 0.4 after 32 blocks. This contraction is related to the *rank collapse* that Dong et al. (2021) prove for stacks of attention layers without residual connections, where the outputs converge to a single token with depth.

Collapsed tokens starve attention of gradient. When every token is nearly the same vector, every query scores every key alike, so the attention weights, and with them the loss, barely depend on $\mathbf{W}_q$ and $\mathbf{W}_k$. At block 32 the post-LN query and key projections receive gradients of order 10^{-9}, five to six orders of magnitude less than in the pre-LN stack, while the value and output projections, which act on the averaged token itself, still receive gradients of several hundredths. These are measurements at initialization, but they agree with what Xiong et al. (2020) find in training: post-LN needs a learning-rate warmup to train well, and pre-LN trains without one. We use pre-norm as the default and compare both placements in training in Section 11.2.

11.1.2.3 RMSNorm

Modern models also simplified the normalizer itself. LayerNorm centers each token vector, divides it by its standard deviation, and then scales and shifts it. *RMSNorm* (Zhang and Sennrich 2019) drops the centering and the shift and keeps only the part that controls the scale:

$$\text{RMSNorm}(\mathbf{x}) = \frac{\mathbf{x}}{\sqrt{\frac{1}{d}\sum_{i=1}^{d} x_i^2 + \epsilon}} \odot \gamma. \tag{11.3}$$

It is a one-line function, and both frameworks ship it. We check our implementation against the built-in and time the two normalizers with `d2l.Benchmark`, which Section 13.1 introduces: it calls a function a few times to warm up, then reports the average time of further calls, waiting for the device to finish before it stops the clock.

```
def rms_norm(X, weight, eps=1e-5):
    return X / torch.sqrt((X * X).mean(-1, keepdim=True) + eps) * weight

device = d2l.try_gpu()
X = torch.randn(64, 512, 1024, device=device)
layernorm = nn.LayerNorm(1024, eps=1e-5, device=device)
rmsnorm = nn.RMSNorm(1024, eps=1e-5, device=device)
print(f'ours vs. built-in: max deviation '
      f'{(rms_norm(X, rmsnorm.weight) - rmsnorm(X)).abs().max():.1e}')
for name, f in (('LayerNorm', layernorm), ('RMSNorm', rmsnorm)):
    print(d2l.Benchmark(lambda: f(X), warmup=10, repeats=200, desc=name))
```

```
ours vs. built-in: max deviation 9.5e-07
LayerNorm: 0.72 ms/call
RMSNorm: 0.72 ms/call
```

Our implementation agrees with the built-in to rounding error. RMSNorm computes one statistic per token instead of two and has no shift parameter. How much time this saves depends on the kernel more than on the formula. Both normalizers read and write the whole tensor, and when that memory traffic dominates, as it often does on GPUs, the saved arithmetic changes little. The two timings above therefore vary with the framework and the device. RMSNorm is preferred because it does less work at equal quality. Zhang and Sennrich (2019) match LayerNorm's accuracy across several tasks while reducing the running time of their models by 7% to 64%, and Llama adopted it (Touvron et al. 2023a).

11.1.2.4 Normalizing Queries and Keys

A further normalization sits *inside* attention. Nothing in the block above controls the size of the attention logits $\mathbf{q}^\top \mathbf{k}/\sqrt{p}$, where p is the head width. If training inflates the queries and keys, the softmax saturates, its Jacobian vanishes (Section 10.2), and training can spike or stall. The instability was first reported in a 22-billion-parameter vision transformer, whose attention logits grew until the attention weights were nearly one-hot (Dehghani et al. 2023), and Wortsman et al. (2024) reproduce it in small language models trained at high learning rates. *QK-norm* (Henry et al. 2020) applies RMSNorm to the queries and the keys of each head right before the dot product, so the size of the logits no longer grows with the query and key weights. It adds two small norms per layer, and both studies find that it removes the instability.

Recent open models also place norms on the other side of a sublayer. OLMo 2 normalizes what each sublayer *writes* before adding it to the stream (Team OLMo et al. 2025), and Gemma 3 normalizes each branch both before and after the sublayer (Gemma Team 2025); both also use QK-norm. Figure 11.2 compares the four placements. Only post-LN normalizes the stream itself; the signal experiment above compared it with pre-LN. QK-norm is not

a flag of our block. It is an exercise, and it enters the block through the `attn_factory` hook introduced below.

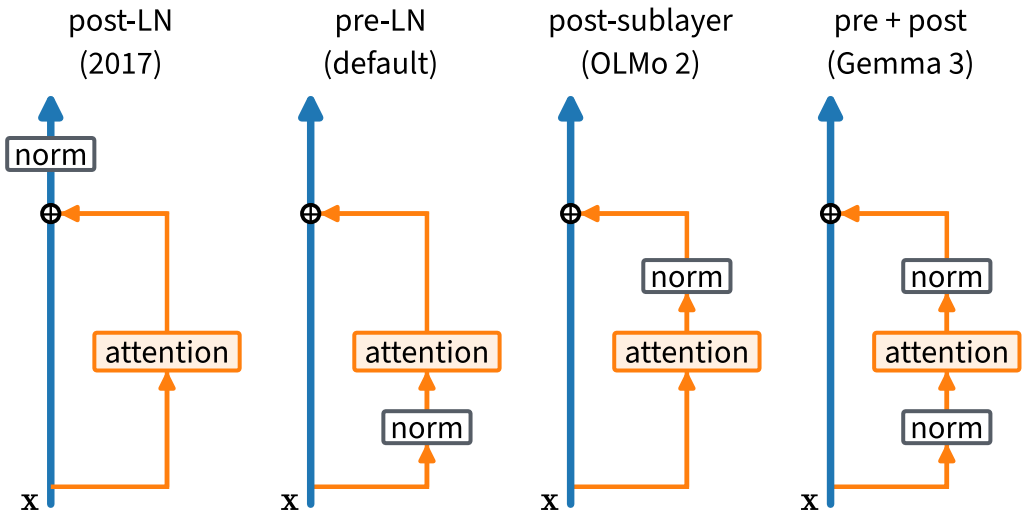


Figure 11.2: Four placements of normalization around one sublayer, with the residual stream running bottom to top; the FFN sublayer is treated the same way. Post-LN normalizes the stream itself after each addition, pre-LN normalizes what the branch reads, OLMo 2 normalizes what the branch writes, and Gemma 3 normalizes the branch on both sides. Only post-LN touches the stream's identity path.

11.1.3 The Feed-Forward Network

Attention moves information between positions but computes no nonlinear feature of what a single position holds. That computation happens in the other half of the block, a small MLP applied to each position independently,

$$\mathrm{FFN}(\mathbf{x}) = \mathbf{W}_2\,\phi(\mathbf{W}_1\mathbf{x}), \tag{11.4}$$

with $\mathbf{W}_1 \in \mathbb{R}^{4d\times d}$ and $\mathbf{W}_2 \in \mathbb{R}^{d\times 4d}$ in the classic configuration. The expansion by a factor of 4 dates from the 2017 transformer and is still the common default. With it, the FFN holds $8d^2$ parameters against the $4d^2$ of the four attention projections $\mathbf{W}_q, \mathbf{W}_k, \mathbf{W}_v, \mathbf{W}_o$, so about two thirds of the parameters of a block sit in these two matrices. Geva et al. (2021) read them as a memory with the structure of attention: each row of $\mathbf{W}_1$ acts as a key that responds to a pattern in the token's vector, and the matching column of $\mathbf{W}_2$ is the value that the FFN adds to the stream when its key responds.

Two changes separate the 2017 FFN from current ones. First, the activation ϕ moved from ReLU to the *GELU* $\phi(z) = z\,\Phi(z)$ of Section 4.1 (Hendrycks and Gimpel 2016), where Φ is the standard normal cumulative distribution function. GPT used it from the start (Radford et al. 2018). Second, Shazeer (2020) revived an older idea, gating, and found that a *gated linear unit* with the SiLU activation beat both:

$$\text{SwiGLU}(\mathbf{x}) = \mathbf{W}_2\big(\text{SiLU}(\mathbf{W}_g\mathbf{x}) \odot \mathbf{W}_1\mathbf{x}\big), \qquad \text{SiLU}(z) = z \cdot \text{sigmoid}(z). \tag{11.5}$$

Instead of passing a linear transform through a fixed nonlinearity, SwiGLU computes *two* linear views of the input and multiplies them. One view is squashed into a soft gate. The other passes through unchanged, so wherever the gate is open it is transmitted at full strength instead of being bent by the activation. Gates in LSTMs (Section 12.1) use the same multiplicative interaction. A third matrix means more parameters at the same width, so fair comparisons shrink the hidden width to $\frac{8}{3}d$. The three matrices of the gated FFN then hold the same $8d^2$ parameters as the classic two, up to rounding. Both variants fit in one class:

```
class FeedForward(nn.Module):
    """Position-wise FFN: GELU MLP or SwiGLU at matched parameter count."""
    def __init__(self, num_hiddens, act='swiglu', bias=False):
        super().__init__()
        assert act in ('swiglu', 'gelu'), f'unknown act: {act!r}'
        self.act = act
        if act == 'gelu':
            width = 4 * num_hiddens
        else:  # 'swiglu': three matrices; width 8d/3 matches the MLP budget
            width = round(8 * num_hiddens / 3)
            self.W_g = nn.Linear(num_hiddens, width, bias=bias)
        self.W_1 = nn.Linear(num_hiddens, width, bias=bias)
        self.W_2 = nn.Linear(width, num_hiddens, bias=bias)

    def forward(self, X):
        if self.act == 'gelu':  # The tanh approximation, as in GPT-2
            return self.W_2(F.gelu(self.W_1(X), approximate='tanh'))
        return self.W_2(F.silu(self.W_g(X)) * self.W_1(X))
```

Both tabs use the tanh approximation of GELU, the variant GPT-2 was trained with, which matters when we load its weights in Section 11.2. We check the parameter match with `d2l.num_params` from Section 7.5, which counts the trainable parameters of a model:

```
for act in ('gelu', 'swiglu'):
    ffn = FeedForward(256, act)
    print(f'{act:>7}: {d2l.num_params(ffn)} parameters')
```

```
   gelu: 524288 parameters
 swiglu: 524544 parameters
```

The two variants differ by 256 parameters, 0.05% of the total. Whether the gate improves the model at equal size is an empirical question, and the last experiment of this section answers it in a small language model.

11.1.4 A Configurable Block

We now assemble the block used throughout this chapter. Every design decision above becomes a constructor argument: `norm` selects LayerNorm or RMSNorm, `act` selects the FFN, and `pre_norm` selects the arrangement. Two factory hooks, `attn_factory` and `ffn_factory`, let

later sections swap whole sublayers without touching this class again: a causal attention, an attention with a smaller key–value cache, or a mixture-of-experts FFN. The constructor checks its flag strings, so a typo raises an error instead of silently selecting another architecture. The default configuration (pre-norm, RMSNorm, SwiGLU) is the one Llama adopted in 2023 (Touvron et al. 2023a), and many open-weights decoders have kept it since. `TransformerBlock(num_hiddens, num_heads, norm='layer', act='gelu', pre_norm=False, bias=True)` is a compact post-norm block in the 2017 encoder arrangement, with one anachronism kept for simplicity: GELU, where the original used ReLU.

```
class TransformerBlock(nn.Module):
    """Configurable transformer block: attention + FFN on a residual
    stream."""
    def __init__(self, num_hiddens, num_heads, *, dropout=0, norm='rms',
                 act='swiglu', pre_norm=True, bias=False, attn_factory=None,
                 ffn_factory=None):
        super().__init__()
        assert norm in ('rms', 'layer'), f'unknown norm: {norm!r}'
        assert num_hiddens % num_heads == 0
        self.pre_norm = pre_norm
        make_norm = nn.RMSNorm if norm == 'rms' else nn.LayerNorm
        self.norm1 = make_norm(num_hiddens, eps=1e-5)
        self.norm2 = make_norm(num_hiddens, eps=1e-5)
        self.attention = (d2l.MultiHeadAttention(num_hiddens, num_heads,
                                                 dropout, bias=bias)
                          if attn_factory is None else attn_factory())
        self.ffn = (FeedForward(num_hiddens, act, bias=bias)
                    if ffn_factory is None else ffn_factory())
        self.dropout = nn.Dropout(dropout)

    def forward(self, X, mask=None, **attn_args):
        # attn_args go on to the attention, e.g. the RoPE tables of d2l.GPT
        if self.pre_norm:
            Y = self.norm1(X)
            X = X + self.dropout(self.attention(Y, Y, Y, mask, **attn_args))
            return X + self.dropout(self.ffn(self.norm2(X)))
        X = self.norm1(X + self.dropout(self.attention(X, X, X, mask,
                                                       **attn_args)))
        return self.norm2(X + self.dropout(self.ffn(X)))
```

11.1.4.1 Shapes and Parameters

Whatever the flags, the block preserves the shape of its input. Its parameter count confirms the accounting above: $4d^2$ in attention, $8d^2$ in the FFN however the activation divides it, and a few normalization weights. That makes about $12d^2$ per block, two thirds of it in the FFN.

```
X = torch.ones(2, 10, 256)
blk = TransformerBlock(256, num_heads=8)
blk2017 = TransformerBlock(256, num_heads=8, norm='layer', act='gelu',
                           pre_norm=False, bias=True)
d2l.check_shape(blk(X), X.shape)
```

```
d2l.check_shape(blk2017(X), X.shape)
attn, ffn, total = (d2l.num_params(m) for m in (blk.attention, blk.ffn, blk))
print(f'attention {attn}, ffn {ffn}, total {total} = '
      f'{total / 256 ** 2:.2f} d^2')
```

```
attention 262144, ffn 524544, total 787200 = 12.01 d^2
```

At $d = 256$ the block holds $12.01d^2$ parameters. The same count scales to GPT-2 small, whose blocks have width $d = 768$: $12d^2 \approx 7.1$ million parameters per block, and about 85 million in its 12 blocks. Its token and position embeddings add another 39 million, which brings the model to its 124 million parameters.

One block is not yet a language model, but a few lines turn a stack of blocks into one we can train. A boolean mask makes the block *causal*. The mask `d2l.causal_mask(T)` (Section 10.2) lets the query at position t see the keys at positions $0, \dots, t$, so every token attends only backward. A dozen lines wrap an embedding, a stack of blocks, a final RMSNorm, and a tied output head (Section 10.4) around the mask:

```
class CharLM(nn.Module):
    """Minimal byte-level LM: causal masking through a boolean mask."""
    def __init__(self, vocab_size, num_hiddens=128, num_heads=4, num_blks=2,
                 act='swiglu', max_len=128):
        super().__init__()
        self.token_emb = nn.Embedding(vocab_size, num_hiddens)
        self.pos_emb = nn.Embedding(max_len, num_hiddens)
        for emb in (self.token_emb, self.pos_emb):
            nn.init.normal_(emb.weight, std=0.02)
        self.blks = nn.ModuleList([
            TransformerBlock(num_hiddens, num_heads, act=act)
            for _ in range(num_blks)])
        self.norm = nn.RMSNorm(num_hiddens, eps=1e-5)

    def forward(self, X):
        T = X.shape[1]
        H = self.token_emb(X) + self.pos_emb(
            torch.arange(T, device=X.device))
        causal = d2l.causal_mask(T, device=X.device)
        for blk in self.blks:
            H = blk(H, causal)
        return F.linear(self.norm(H), self.token_emb.weight)
```

11.1.4.2 Comparing GELU and SwiGLU

SwiGLU keeps its parameter count equal to the GELU network's by shrinking the hidden width, so a comparison of the two asks whether the gate makes a better model at the same size. We train `CharLM` twice on `d2l.WikiText103`, the bytes of the Wikipedia articles introduced in Section 9.5.2. Both runs use the same model size, optimizer, learning rate, and budget of 600 steps. Only the `act` flag differs, and the parameter counts match to a tenth of a percent. Each line reports the validation loss after 200, 400, and 600 steps, measured on the held-out articles with `d2l.lm_perplexity` (Section 9.5), whose logarithm is the mean loss in nats per byte.

Two shorthands keep the experiment the same in both frameworks: `charlm` builds a `CharLM` with fresh random weights, and `adamw` returns AdamW without weight decay.

```
charlm = CharLM

def adamw(model, lr):
    """Plain AdamW without weight decay."""
    return torch.optim.AdamW(model.parameters(), lr=lr, weight_decay=0.0)
```

```
data = d2l.WikiText103(batch_size=64, num_steps=128)
for act in ('gelu', 'swiglu'):
    model = charlm(len(data.vocab), act=act)
    optimizer = adamw(model, lr=1e-3)
    val = []
    for _ in range(3):  # Three rounds of 200 steps
        d2l.train_steps(model, data, optimizer, 200)
        val.append(math.log(d2l.lm_perplexity(model, data.val_dataloader())))
    print(f'{act:>7}: validation loss at step 200/400/600: '
          + '/'.join(f'{v:.2f}' for v in val))
```

```
   gelu: validation loss at step 200/400/600: 2.49/2.16/1.89
 swiglu: validation loss at step 200/400/600: 2.47/2.21/1.88
```

The gated FFN ends a few hundredths of a nat ahead, and it stays ahead when we repeat the comparison from fresh random initializations. The direction agrees with the comparison of Shazeer (2020) at the scale of T5. There, at matched parameters and computation, SwiGLU lowered the held-out log-perplexity by about 0.04 nats against GELU. It also agrees with the choice of the major open model families since Llama. The improvement at equal size is modest, and a modern block combines several improvements of this kind.

We can now build a transformer block in any of the configurations used in practice and predict how it treats signals at initialization. `CharLM` is still a toy: two blocks, learned absolute positions, and 600 steps of training. The next section builds a GPT from the same blocks, with rotary positions, trains it as a language model, and tests the placement of normalization in training.

11.1.5 Summary

Attention moves information between positions but computes no nonlinear feature of any one position. A transformer block therefore pairs it with a position-wise FFN, which holds about two thirds of the parameters of the block, and both add their outputs to a residual stream. In the original post-LN arrangement, normalization pins the scale of the stream. At initialization a 32-block stack then collapses its tokens geometrically, and the query and key projections of the top block receive gradients five to six orders of magnitude smaller than in the pre-norm stack. Pre-norm normalizes what each branch reads, lets the stream grow like the square root of the depth, and keeps every block trainable. RMSNorm keeps only the scale statistic of LayerNorm, and QK-norm bounds the attention logits. SwiGLU replaces the fixed activation with a learned gate; at matched parameter count it ends a few hundredths of a nat ahead of GELU in our byte-level model. The `TransformerBlock` class exposes these choices as options, with replaceable attention and FFN modules.

11.1.6 Exercises

1. **Scale invariance.** Let RMSNorm act as in Equation 11.3 with $\gamma = \mathbf{1}$ and $\epsilon = 0$, and write $r(\mathbf{x}) = (\frac{1}{d}\sum_i x_i^2)^{1/2}$ for the root mean square of $\mathbf{x} \in \mathbb{R}^d$.
   1. Show that $\mathrm{RMSNorm}(\alpha\mathbf{x}) = \mathrm{RMSNorm}(\mathbf{x})$ for every $\alpha > 0$, and that LayerNorm with unit gain and zero shift is also unchanged when the same constant is added to every coordinate. Describe the set of inputs that each normalizer maps to a given output.
   2. Show that the Jacobian of RMSNorm at $\mathbf{x}$ is $r(\mathbf{x})^{-1}(\mathbf{I} - \hat{\mathbf{x}}\hat{\mathbf{x}}^\top)$ with $\hat{\mathbf{x}} = \mathbf{x}/\|\mathbf{x}\|$. Conclude that the gradient $\nabla_{\mathbf{x}}\ell$ is orthogonal to $\mathbf{x}$ for every loss ℓ that depends on $\mathbf{x}$ only through $\mathrm{RMSNorm}(\mathbf{x})$.
2. **Unrolled residual stream.** Number the $2N$ sublayers of an N-block stack by $j = 1, \dots, 2N$, write F_j for the j-th sublayer (attention or FFN) and Norm_j for its normalization, and let $\mathbf{x}_j$ be the residual stream after sublayer j, so that $\mathbf{x}_0$ is the input.
   1. For the pre-LN arrangement Equation 11.1, write $\mathbf{x}_{2N}$ as $\mathbf{x}_0$ plus a sum of $2N$ branch outputs. Write the Jacobian $\partial\mathbf{x}_{2N}/\partial\mathbf{x}_0$ as a product over sublayers, expand the product, and identify the term that passes a gradient from the output to the input unchanged.
   2. For the post-LN arrangement Equation 11.2, the stream is $\mathbf{x}_j = \mathrm{Norm}_j(\mathbf{u}_j)$ with $\mathbf{u}_j = \mathbf{x}_{j-1} + F_j(\mathbf{x}_{j-1})$. Model every normalization as RMSNorm with unit gain and $\epsilon = 0$. Assume that the input $\mathbf{x}_0$ has unit root mean square and that each branch output is uncorrelated with the stream it is added to. The product of the normalization Jacobians alone takes the place of the identity term of the pre-LN expansion. Show that this product scales a vector orthogonal to every $\mathbf{u}_j$ by $\prod_j r(\mathbf{u}_j)^{-1}$, and write this factor as a function of the mean squares $c_1, \dots, c_{2N}$ of the branch outputs. Evaluate it for a common mean square $c_j = c$, and compare its dependence on N with that of the identity term of the pre-LN expansion.
   3. `CharLM` applies a final RMSNorm, `norm`, before its tied output head. Describe what the head would receive without it as a pre-LN stack grows deeper, and explain why a post-LN stack needs no final normalization.
3. code **QK-norm as a factory.** QK-norm (Section 11.1.2.4) applies RMSNorm over the head dimension $p = d/h$ to every query and key just before the scores are computed.
   1. Write `QKNormAttention`, a subclass of `d2l.MultiHeadAttention` whose `forward` (`__call__` in JAX) passes the per-head queries and keys produced by `split_heads` through their own RMSNorm each before the dot product. Build a `TransformerBlock` with it through `attn_factory` and check the output shape.
   2. Show that with unit gains every scaled logit $\hat{\mathbf{q}}^\top\hat{\mathbf{k}}/\sqrt{p}$ lies in $[-\sqrt{p}, \sqrt{p}]$, and generalize the bound to learned gains γ_q and γ_k. Compute the largest possible ratio between two attention weights of one query for $p = 64$ and unit gains.
   3. Training can inflate the query and key weights. Mimic this at initialization: multiply the weights of `W_q` and `W_k` of a freshly initialized layer with $d = 256$ and $h = 4$ by $\lambda \in \{1, 2, 4, 8\}$, and feed it 64 standard normal tokens. Measure the standard deviation of the logits and the mean entropy of the attention weights, with and without QK-norm. Compare the entropies with $\log 64$, and explain the shape of each curve.
4. **FFN width rounding.** `FeedForward` sets the SwiGLU width to $\mathrm{round}(8d/3)$.
   1. Write the exact parameter count of the default `TransformerBlock(d, h)` as a function of d, and bound the relative deviation of its FFN from $8d^2$.
   2. The released 7B configuration of Llama (Touvron et al. 2023a) has $d = 4096$ and

SwiGLU width 11008, the smallest multiple of 256 that is at least $\lfloor 8d/3 \rfloor$. Compute the relative excess of its FFN parameters over $8d^2$, and the excess that the same rule gives at $d = 256$ and at $d = 128$, the default `num_hiddens` of `CharLM`. Explain what the comparison implies for parameter-matched experiments with small models.

5. code **Causal masks.** `CharLM` passes the boolean mask `causal = d2l.causal_mask(T)` to every block, so that the query at position $t = 0, \dots, T-1$ attends to the keys at positions $0, \dots, t$.
   1. Show by induction over the blocks that the logits of `CharLM` at position t depend only on the tokens at positions $0, \dots, t$. Which step of the induction fails if only the first block receives `causal` and the other blocks receive no mask?
   2. Give `CharLM` an argument `shift` and build the mask as `torch.ones(T, T, dtype=torch.bool).tril(shift)` (PyTorch) or `jnp.tril(jnp.ones((T, T), bool), shift)` (JAX), so that the query at position t sees the keys at positions $0, \dots, t + s$, where s is the value of `shift`. For `shift=-1`, the query at position 0 sees no key, and `d2l.masked_softmax` gives it zero weights. For `shift=1` and for `shift=-1`, predict the positions whose logits depend on the token at position j in a stack of N blocks. Which of the two settings can be trained as a next-token predictor?
   3. Check your predictions on freshly initialized models with `num_blks` $\in \{1, 2, 4\}$: change the token at position 16 of a sequence of 32 random tokens, and find the positions whose logits change.
6. code **Stream variance budget.** In the pre-LN arrangement every branch reads a normalized input. If each branch output is uncorrelated with the stream it is added to, the mean square of the stream after k blocks is

$$1 + \sum_{i=1}^{k} (c_i^{\text{attn}} + c_i^{\text{ffn}}),$$

   where c_i^{attn} and c_i^{ffn} are the mean squares of the attention and FFN outputs of block i. Use `MiniBlock` with the Xavier initialization of `signal_stats` at $d = 256$.
   1. Predict c_1^{ffn}. Compute the variance of the hidden pre-activations of the FFN, and evaluate the second moment of GELU at that variance numerically.
   2. Predict c_1^{attn}. Treat the $n = 64$ input tokens as independent, and treat the attention logits of one query as independent Gaussians with the variance that the initialization implies.
   3. Record the mean square of each branch output in `signal_stats` and plot c_k^{attn} and c_k^{ffn} against k. Compare block 1 with your predictions, and explain the trend of c_k^{attn} using the token spread.
   4. Fit a power law to the stream RMS, once over blocks 1 to 32 and once over blocks 8 to 32. Compare both exponents with $1/2$, explain the comparison with the variance budget, and state the regime in which the square-root growth stated in Section 11.1.2.2 holds.
7. code **Rate of token collapse.** Write the tokens of one sequence as $\mathbf{x}_i = \mu + \delta_i$ for $i = 1, \dots, n$, where μ is the mean token and $\sum_i \delta_i = \mathbf{0}$. With the ratio of common to deviation energy

$$\rho = \frac{n\|\mu\|^2}{\sum_i \|\delta_i\|^2}, \tag{11.6}$$

   the token spread of `signal_stats` equals $(1 + \rho)^{-1/2}$. At initialization the attention weights are nearly uniform, so a post-LN attention sublayer adds approximately the

same vector $\mathbf{W}_o\mathbf{W}_v\mu$ to every token, where $\mathbf{W}_v, \mathbf{W}_o \in \mathbb{R}^{d\times d}$ are the value and output projections.

1. Let the entries of $\mathbf{W}_v$ and $\mathbf{W}_o$ be independent with mean zero and variance α^2/d, so that $\alpha = 1$ is Xavier initialization of a square matrix. Treat the normalization as a rescaling common to all tokens. Show that one attention sublayer and its normalization multiply ρ by about $1 + \alpha^4$, and derive the spread after k blocks for an input of n independent random tokens, ignoring the FFN sublayers.
2. Rerun `signal_stats` for post-LN with all Xavier weights multiplied by $\alpha \in \{1, 0.7, 0.5\}$, once with the full `MiniBlock` and once with its FFN sublayer removed. Compare both with your formula at blocks 8, 16, and 32, and explain the remaining gap of the full block.
3. GPT-1 stacked 12 post-LN blocks of width $d = 768$ and drew every weight from $\mathcal{N}(0, 0.02^2)$ (Radford et al. 2018). Compute α for its attention projections, and predict the spread after 12 blocks for $n = 64$, both for this initialization and for Xavier initialization at the same width. Check both predictions with `signal_stats`.

8. [extended] **Learning-rate sensitivity.** ⊠ Wortsman et al. (2024) reproduce the growth of attention logits reported for large training runs (Section 11.1.2.4) in small models trained at high learning rates, and find that QK-norm removes it. They summarize the final loss $\ell(\eta)$ as a function of the learning rate η by the *learning-rate sensitivity*

$$\mathbb{E}_\eta\big[\min(\ell(\eta), \ell_0) - \ell^*\big], \tag{11.7}$$

where ℓ_0 is the loss at initialization, ℓ^* is the best final loss over the learning rates tried, and the average runs over learning rates spanning three orders of magnitude.

1. Give `CharLM` an option that builds its blocks with `QKNormAttention` through `attn_factory`. With `num_hiddens=256` and `num_blks=6`, train it with and without QK-norm for 1000 steps at each learning rate in $\{10^{-4}, 10^{-3.5}, \dots, 10^{-1}\}$, otherwise under the protocol of Section 11.1.4.2.
2. Every 100 steps, record the training loss and the largest absolute attention logit on a fixed batch. Plot the final loss against η for both models, and compute the learning-rate sensitivity of each curve.
3. For every learning rate at which the model without QK-norm ends worse than the model with it, decide from the recorded attention logits whether attention-logit growth explains the gap. If it does not, record the largest absolute output logit of both models at the same steps, and decide whether its growth explains the gap.

11.2 A GPT from Scratch

The block of Section 11.1 maps a sequence of vectors to a sequence of the same shape. A language model has a more specific job: it reads text and says what comes next. How does a stack of blocks become such a model, and how do we train it?

The training text suggests the answer. A language model solves the same prediction problem at every position of every training sequence. Given "the European lobste", it must predict "r"; given "the European lobster", it must predict a space. One sequence of 128 bytes therefore holds 128 training examples, and a model can learn from all of them at once if it meets two requirements. Each position must predict its successor without reading it or any later token. And the same parameters must serve every position and every length of context, from one token to the whole window.

The transformer block meets the second requirement already: it applies the same weights at

every position. A causal mask inside attention meets the first. Adding the remaining parts, an embedding, positions, and an output head, gives a GPT (*generative pretrained transformer*), which we write as a `GPT` class. We train it on the bytes of Wikipedia articles, test where the normalization belongs, and sample text from it. Finally, we load the released 124-million-parameter GPT-2 (Radford et al. 2019) into the same class, which the rest of this chapter reuses. Its sampling loop recomputes the whole context for every new token, a waste that the next section removes.

11.2.1 From Blocks to a Language Model

Four ingredients turn a stack of transformer blocks into a language model, and each solves one part of the problem. Figure 11.3 shows how they fit together.

A token embedding and a tied head. A table $\mathbf{E}$ with one row of width d per vocabulary entry maps token ids into the residual stream. After the final block and one last normalization, the *same* table maps the stream back: the logit of token v is the dot product of the stream with row v of $\mathbf{E}$, so all logits together are $\mathbf{Eh}$. Sharing, or *tying*, the two matrices saves $V \times d$ parameters for a vocabulary of size V and helps at small scale (Press and Wolf 2017). GPT-2 ties them too, which spares us a tensor when we load its weights.

A causal mask. Position t predicts token $t + 1$ from tokens $1, \dots, t$, so its query may attend only to keys at positions up to t. In Section 11.1 the small language model passed this lower-triangular mask to every block as a boolean array. Here the attention module applies it itself: it passes the attention function the flag `is_causal=True` instead of a mask. The fused attention kernels of Section 10.5 take the causal mask in this form and never store the $n \times n$ matrix of scores. With the mask, one forward pass over a window computes all its next-token predictions at once, each from its own prefix.

Positions. Attention alone treats its inputs as a set: permuting the tokens permutes the outputs in the same way (Section 10.4). The `pos` argument selects one of the two schemes of that section. `'learned'` adds a trained vector per position to the embedding, as GPT-2 does, which limits the context to the length of the table. `'rope'` rotates the queries and keys of every head by an angle proportional to their position, so that the score of a query and a key depends on their offset. The attention module calls `d2l.rope` where it computes the queries and keys. The angles depend only on the positions, so `GPT` computes their cosines and sines once per forward pass with `d2l.rope_tables` and passes them to every block.

The attention module below puts the mask and the positions in one place. One matrix `W_qkv` computes the queries, keys, and values together; `qkv` splits them into heads and, for RoPE, rotates the queries and the keys; and the attention call is causal. A block passes its attention keys, values, and an optional mask, and any keyword arguments of its own call, here the RoPE tables. This module uses its input as the keys and the values, and it rejects a mask, because its own mask is always causal.

```
class CausalAttention(nn.Module):
    """Fused causal self-attention, optionally rotary; num_kv_heads <
    num_heads shares each key-value head across a group of queries."""
    def __init__(self, num_hiddens, num_heads, *, bias=False, rope=False,
                 num_kv_heads=None):
        super().__init__()
        self.num_heads, self.rope = num_heads, rope
```

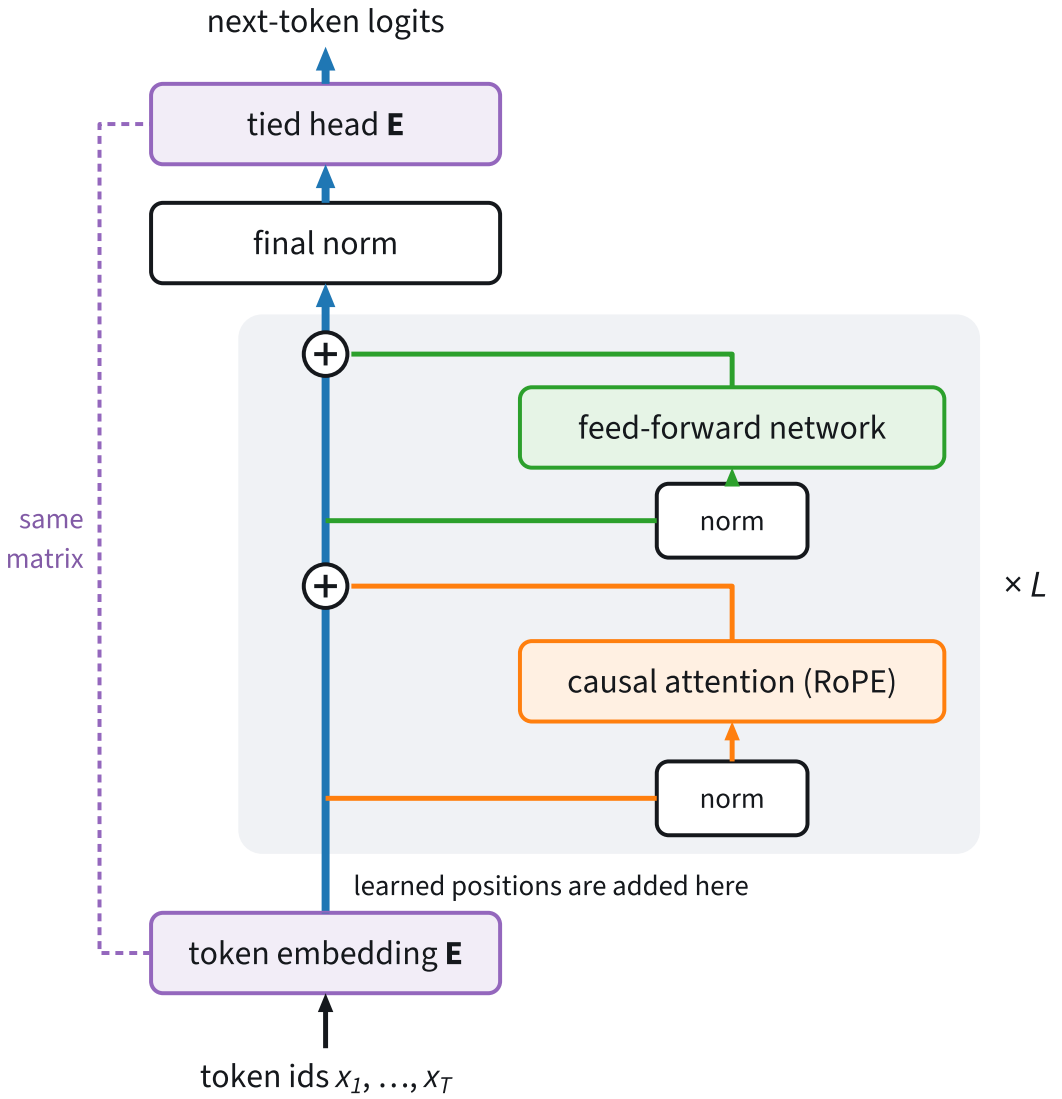


Figure 11.3: A GPT from bottom to top. A table **E** embeds the token ids into the residual stream, L pre-norm blocks of causal attention and feed-forward network add to the stream, and after a final normalization the same table **E** maps the stream back to one logit per vocabulary entry. Learned positions are added after the embedding; RoPE instead rotates queries and keys inside each attention layer.

```
        self.num_kv_heads = num_kv_heads or num_heads
        self.kv_hiddens = self.num_kv_heads * (num_hiddens // num_heads)
        self.W_qkv = nn.Linear(num_hiddens, num_hiddens + 2 * self.kv_hiddens,
                               bias=bias)
        self.W_o = nn.Linear(num_hiddens, num_hiddens, bias=bias)

    def qkv(self, X, offset=0, tables=None):
        """Queries, keys, and values of X, each (batch, heads, length, head
        width), at positions offset, offset + 1, ...; tables, if given, are
        the d2l.rope_tables of these positions."""
        B, T, D = X.shape
        q, k, v = self.W_qkv(X).split([D, self.kv_hiddens, self.kv_hiddens],
                                      -1)
        q = q.reshape(B, T, self.num_heads, -1).transpose(1, 2)
        k, v = (u.reshape(B, T, self.num_kv_heads, -1).transpose(1, 2)
                for u in (k, v))
        if self.rope:
            q, k = (d2l.rope(u, offset, tables=tables) for u in (q, k))
        return q, k, v

    def forward(self, X, keys=None, values=None, mask=None, tables=None):
        # Self-attention: X also serves as the keys and the values, and the
        # mask is causal, so a block's mask argument must stay None
        if mask is not None:
            raise ValueError('CausalAttention applies its own mask')
        q, k, v = self.qkv(X, tables=tables)
        Y = F.scaled_dot_product_attention(
            q, k, v, is_causal=True,
            enable_gqa=self.num_kv_heads < self.num_heads)
        return self.W_o(Y.transpose(1, 2).flatten(2))  # (batch, length, d)
```

The blocks. `d2l.TransformerBlock` is used unchanged, with its defaults of Section 11.1: pre-norm, RMSNorm, and a SwiGLU feed-forward network. The causal attention enters through its `attn_factory` argument. Its option `num_kv_heads` lets groups of query heads share one key–value head, which later sections use to make generation cheaper.

`GPT` keeps the attention module as its class attribute `CausalAttention`, so that a subclass can replace it, as Section 13.7 does to select a faster kernel.

```
class GPT(nn.Module):
    """Decoder-only transformer language model built from configurable
    blocks."""
    CausalAttention = CausalAttention  # A subclass may replace it

    def __init__(self, vocab_size, *, num_hiddens=256, num_heads=8,
                 num_blks=6, max_len=1024, pos='rope', norm='rms',
                 act='swiglu', pre_norm=True, bias=False, dropout=0,
                 num_kv_heads=None, ffn_factory=None):
        super().__init__()
        assert pos in ('rope', 'learned'), f'unknown pos: {pos!r}'
        self.pos, self.max_len, self.num_hiddens = pos, max_len, num_hiddens
        self.head_width = num_hiddens // num_heads
```

```
        self.token_emb = nn.Embedding(vocab_size, num_hiddens)
        nn.init.normal_(self.token_emb.weight, std=0.02)
        if pos == 'learned':
            self.pos_emb = nn.Embedding(max_len, num_hiddens)
            nn.init.normal_(self.pos_emb.weight, std=0.02)
        attn = lambda: self.CausalAttention(
            num_hiddens, num_heads, bias=bias, rope=pos == 'rope',
            num_kv_heads=num_kv_heads)
        self.blks = nn.ModuleList([
            d2l.TransformerBlock(num_hiddens, num_heads, dropout=dropout,
                                 norm=norm, act=act, pre_norm=pre_norm,
                                 bias=bias, attn_factory=attn,
                                 ffn_factory=ffn_factory)
            for _ in range(num_blks)])
        self.norm = (nn.RMSNorm if norm == 'rms'
                     else nn.LayerNorm)(num_hiddens, eps=1e-5)
        self.init_weights()

    def forward(self, X):                       # X: (batch, T) token ids
        H, tables = self.token_emb(X), None     # (batch, T, d)
        if self.pos == 'learned':
            H = H + self.pos_emb(torch.arange(X.shape[1], device=X.device))
        else:  # RoPE angles of positions 0, ..., T - 1, for every block
            tables = d2l.rope_tables(X.shape[1], self.head_width,
                                     device=X.device)
        for blk in self.blks:                   # L blocks, causal attention
            H = blk(H, tables=tables)
        return F.linear(self.norm(H), self.token_emb.weight)  # (batch, T, V)
```

`GPT` initializes its weights as GPT-2 does. Every weight matrix is drawn from $\mathcal{N}(0, 0.02^2)$ and every bias starts at zero, with one exception. Each block has two matrices that write into the residual stream: the output projection $\mathbf{W}_o$ of attention and the second matrix $\mathbf{W}_2$ of the feed-forward network. Their standard deviation is divided by a further $\sqrt{2L}$, where $2L$ counts the residual branches of L blocks.

The reason is the growth of the stream. In a pre-norm block each branch reads the normalized stream, so the size of its output is set by its weights and not by the stream it reads. The stream is the sum of the embedding and all branch outputs. These are roughly uncorrelated, so their mean squares add, and the mean square of the stream grows in proportion to the number of branches. Dividing the standard deviation of each writing matrix by $\sqrt{2L}$ divides the mean square of each branch output by $2L$. The $2L$ contributions together are then about as large as a single unscaled one, whatever the depth. Without normalization the growth would compound instead, as Section 5.3 showed. This depth scaling is a common choice rather than a law: OLMo 2 returned to a truncated $\mathcal{N}(0, 0.02^2)$ for all parameters (Team OLMo et al. 2025).

The constructor draws the embedding tables this way when it creates them. The method `init_weights`, which it calls last, applies the rule to every linear layer:

```
@d2l.add_to_class(GPT)
def init_weights(self):
```

```
        """GPT-2's initialization of the linear layers: N(0, 0.02^2) weights and
        zero biases; W_o and W_2, which write to the stream, get std
        0.02 / sqrt(2L)."""
        L = len(self.blks)
        for name, m in self.named_modules():
            if isinstance(m, nn.Linear):
                std = 0.02 / (2 * L) ** 0.5 if name.endswith(
                    ('W_o', 'W_2')) else 0.02
                nn.init.normal_(m.weight, std=std)
                if m.bias is not None:
                    nn.init.zeros_(m.bias)
```

A shape check confirms one logit per vocabulary entry at every position, and the standard deviations of two matrices of the first block confirm the initialization:

```
model = GPT(vocab_size=256)
X = torch.zeros(2, 16, dtype=torch.long)
d2l.check_shape(model(X), (2, 16, 256))
attention = model.blks[0].attention
print(f'{d2l.num_params(model) / 1e6:.2f}M parameters; initial std: '
      f'W_qkv {attention.W_qkv.weight.std():.4f}, '
      f'W_o {attention.W_o.weight.std():.4f}')
```

```
4.79M parameters; initial std: W_qkv 0.0199, W_o 0.0058
```

The output projection starts with a standard deviation of $0.02/\sqrt{12} \approx 0.0058$ for the six blocks, the other matrices with 0.02. A forward pass through untrained stacks of increasing depth tests the argument for this scaling. We feed random tokens and measure the mean square of the stream after the last block, once with every matrix at 0.02 and once with the scaled writing matrices:

```
def stream_ms(num_blks, scaled):
    """Mean square of the stream after the last block, for random tokens."""
    net = GPT(vocab_size=256, num_blks=num_blks)
    if not scaled:  # Undo the depth scaling of W_o and W_2
        for name, p in net.named_parameters():
            if name.endswith(('W_o.weight', 'W_2.weight')):
                nn.init.normal_(p, std=0.02)
    with torch.no_grad():
        H = net.token_emb(torch.randint(0, 256, (8, 128)))
        for blk in net.blks:
            H = blk(H)
    return H.square().mean().item()

for scaled in (False, True):
    print(f'scaled={scaled}: mean square after 8/16/32 blocks: ' + '/'.join(
        f'{stream_ms(L, scaled):.4f}' for L in (8, 16, 32)))
```

```
scaled=False: mean square after 8/16/32 blocks: 0.0646/0.1458/0.3192
scaled=True: mean square after 8/16/32 blocks: 0.0026/0.0030/0.0031
```

The embedding alone has a mean square of $0.02^2 = 0.0004$. Without the scaling, the mean

square of the stream about doubles with each doubling of the depth, as a sum of $2L$ comparable terms should. With the scaling, it barely grows from 8 blocks to 32.

The parameter count follows from the block. Attention holds four $d \times d$ matrices, $\mathbf{W}_q$, $\mathbf{W}_k$, $\mathbf{W}_v$, and $\mathbf{W}_o$, and the feed-forward network holds $8d^2$, as two matrices of width $4d$ with GELU or three of width $8d/3$ with SwiGLU. A block thus holds about $12d^2$ parameters, 0.79 million at $d = 256$ and 4.7 million for six blocks, while an embedding of 256 rows adds only 65,536. GPT-2 small has $d = 768$ and 12 blocks, so its blocks hold $12 \times 12 \times 768^2 \approx 85$ million parameters and its token table, $50{,}257 \times 768$, another 38.6 million. An untied head would add a second table of that size and make the model about 31% larger.

11.2.2 Training the Model

11.2.2.1 The Training Text

A model of 4.8 million parameters needs more text than one novel. The training run below makes about 8 million predictions. *The Time Machine* has about 170,000 characters, so the run would pass over the book about fifty times, often enough for a model of this size to memorize much of it rather than learn English (an exercise measures how much). We therefore train on the corpus of the optimization experiments (Section 9.5.2), `d2l.WikiText103`: Wikipedia articles kept as raw text, with case, digits, and punctuation, about 270 million bytes for training. Each training window of 128 bytes starts at a random offset, so the run reads about 3% of the text and almost never the same passage twice. The validation windows are consecutive pieces of the official validation split, about 1.1 million bytes.

```
data = d2l.WikiText103(batch_size=64, num_steps=128)
print(f'{len(data.train) / 1e6:.0f}M training bytes, '
      f'{len(data.val) / 1e6:.2f}M validation bytes, '
      f'vocabulary of {len(data.vocab)}')
```

```
270M training bytes, 1.14M validation bytes, vocabulary of 256
```

The tokens are the UTF-8 bytes of the text, the byte-level tokenizer of Section 8.2 without any merges, rather than its subword tokens. Bytes have two advantages here. The 256-entry table puts 65,536 of the model's parameters into embeddings, about 1.4% of the total, so nearly all of them sit in the blocks; GPT-2's vocabulary of 50,257 tokens would put 12.9 million into the table at width 256, almost three times the rest of the model. And because each token is one byte, the loss in nats per byte divided by $\log 2$ is the number of bits per byte of Section 8.3.2.4, which does not depend on the tokenizer.

11.2.2.2 The Training Recipe

The training recipe is the one Chapter 9 assembled, and each of its four parts addresses a problem met there. The optimizer is AdamW (Section 9.5.4), which divides each coordinate's step by a running estimate of its gradient's magnitude and applies weight decay separately from that rescaling. The decay, $\lambda = 0.1$, acts on the weight matrices only; embeddings, normalization gains, and biases are exempt (Section 9.5.4.5). The learning rate rises linearly from zero over the first 100 steps, while Adam's estimates still rest on few gradients, and then falls along a half cosine to zero at the end of training (`d2l.cosine_warmup`, Equation 7.1). Gradients are clipped to a global norm of 1, which shortens the rare large gradient without changing its direction (Section 9.10).

`LMOptimizer` bundles the four parts behind the interface of an ordinary optimizer, so the training loop `d2l.train_steps` of Section 9.5, which draws minibatches of windows and takes a fixed number of steps on the next-byte loss, uses it unchanged. We use no dropout: a model that almost never reads the same passage twice has nothing to memorize.

```
class LMOptimizer:
    """AdamW with weight decay on the weight matrices only, a linear warmup
    and a cosine decay of the learning rate, and gradient-norm clipping."""
    def __init__(self, model, lr, num_steps, warmup, weight_decay=0.1,
                 max_norm=1.0):
        named = list(model.named_parameters())
        # Embedding tables are named *emb* (token_emb, pos_emb) by convention
        decay = [p for n, p in named if p.ndim == 2 and 'emb' not in n]
        rest = [p for n, p in named if p.ndim != 2 or 'emb' in n]
        self.optimizer = torch.optim.AdamW(
            [{'params': decay, 'weight_decay': weight_decay},
             {'params': rest, 'weight_decay': 0.0}], lr=lr)
        self.scheduler = torch.optim.lr_scheduler.LambdaLR(
            self.optimizer, lambda t: d2l.cosine_warmup(t, num_steps, warmup))
        self.params, self.max_norm = decay + rest, max_norm

    def zero_grad(self):
        self.optimizer.zero_grad()

    def step(self):
        nn.utils.clip_grad_norm_(self.params, self.max_norm)
        self.optimizer.step()
        self.scheduler.step()
```

`LMOptimizer` keeps the two parameter groups of AdamW and a `LambdaLR` scheduler that multiplies the base rate by `d2l.cosine_warmup`. Its `step` clips, updates, and advances the schedule, in that order, so the three always stay in step.

11.2.2.3 Loss Curves and Baselines

A loss in nats means little on its own, so after training we compute reference levels to compare it with. We train for 1,000 steps of 64 windows and record both losses every 50 steps. The training loss is the mean over those 50 steps. The validation loss is the mean cross-entropy per byte on the validation windows, which is the logarithm of `d2l.lm_perplexity` (Section 9.5).

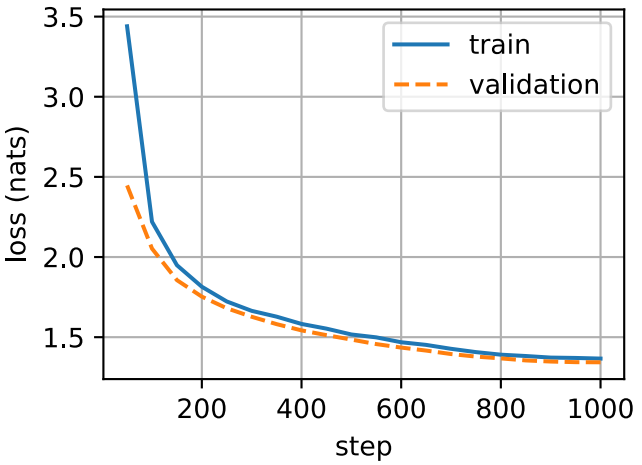
The experiments of this section build their models and plain optimizers through two shorthands: `gpt` constructs a `GPT` with fresh random weights, and `adamw` returns AdamW without weight decay.

```
gpt = GPT

def adamw(model, lr):
    """Plain AdamW without weight decay."""
    return torch.optim.AdamW(model.parameters(), lr=lr, weight_decay=0.0)
```

```
model = gpt(len(data.vocab))
num_steps, every = 1000, 50
optimizer = LMOptimizer(model, lr=2e-3, num_steps=num_steps, warmup=100)
steps, train_curve, val_curve = [], [], []
for step in range(every, num_steps + 1, every):
    losses = d2l.train_steps(model, data, optimizer, every)
    steps.append(step)
    train_curve.append(sum(losses) / every)
    val_curve.append(math.log(d2l.lm_perplexity(model, data.val_dataloader())))
d2l.plot(steps, [train_curve, val_curve], 'step', 'loss (nats)',
         legend=['train', 'validation'], fmts=('-', '--'))
print(f'step {num_steps}: train {train_curve[-1]:.2f}, '
      f'validation {val_curve[-1]:.2f} nats per byte')
```

```
step 1000: train 1.37, validation 1.34 nats per byte
```



The two curves fall together, and the validation loss never turns upward. Almost every training window is text the model has not read before, so what it learns carries over to the held-out articles. At the end, the two losses lie within a few hundredths of a nat of each other. Most of the decrease comes in the first few hundred steps, and both losses end near 1.35 nats per byte, or about 1.9 bits per byte.

Two simple predictors give reference levels for these losses. The *unigram* model predicts every byte with its frequency in the training text and ignores all context. Its cross-entropy on that text is the entropy of the byte frequencies. The *bigram* model predicts each byte from the previous one, with the conditional frequencies of the training text. Both are estimated on the first 10 million bytes of the training text, enough to fix their levels to two decimal places.

```
text = data.train[:10**7].astype(np.int64)  # The first 10M training bytes
V = len(data.vocab)
unigram = np.bincount(text, minlength=V) / len(text)
pairs = np.bincount(text[:-1] * V + text[1:], minlength=V * V).reshape(V, V)
bigram = pairs / np.maximum(pairs.sum(1, keepdims=True), 1)
```

```
p = unigram[unigram > 0]
print(f'unigram {-(p * np.log(p)).sum():.2f}, '
      f'bigram {-np.log(bigram[text[:-1], text[1:]]).mean():.2f} '
      f'nats per byte')
```

```
unigram 3.17, bigram 2.45 nats per byte
```

A model that knows nothing about the text scores $\log 256 \approx 5.55$ nats, close to the loss at initialization. The unigram level is 3.17 nats and the bigram level 2.45. The trained model ends about a nat below the bigram level, so it uses much more of the context than the previous byte.

11.2.2.4 Training Compute

A rough count of operations places this run among larger ones. In the forward pass, every parameter takes part in one multiplication and one addition per token, so a model with N parameters performs about $2N$ floating-point operations per token. The backward pass takes about twice as many. Training on D tokens therefore takes about

$$C \approx 6ND \tag{11.8}$$

operations, ignoring the attention scores, which are small at short contexts. Our run, with $N = 4.8$ million and $D = 8.2$ million, comes to about 2×10^{14}. One pass of GPT-2 small, with 124 million parameters, over roughly ten billion tokens of WebText comes to about 7×10^{18}. The same pass of GPT-2 XL, with 1.5 billion parameters, comes to about 10^{20}, and the largest current training runs exceed 10^{25}. Across these eleven orders of magnitude, the model definition hardly changes: an embedding, a stack of blocks, a causal mask, and a head. What changes is the system around it, from data pipelines to fused kernels and the parallel hardware of Chapter 13.

11.2.2.5 Normalization Placement in Training

Section 11.1 placed the normalization at the input of each sublayer (pre-norm) rather than after each residual addition, as the original transformer did (post-norm). Its argument came from initialization alone. Post-norm rescales the whole stream after every addition. In a deep stack at initialization, this drives the tokens toward their average and leaves the query and key projections with tiny gradients. We now test the placement in training, with the model and data as above. Two runs differ only in `pre_norm`. Both take 800 steps of plain AdamW at a constant learning rate of 10^{-3}, without warmup, a setting in which pre-norm trains without trouble. A third run repeats the post-norm model with a warmup. It uses `LMOptimizer` with the same peak rate, a warmup over the first 400 steps, half the run, and the cosine decay that follows, but without weight decay or clipping.

```
for pre_norm, warmup in ((True, False), (False, False), (False, True)):
    ablation = gpt(len(data.vocab), pre_norm=pre_norm)
    optimizer = (LMOptimizer(ablation, lr=1e-3, num_steps=800, warmup=400,
                             weight_decay=0.0, max_norm=math.inf)
                 if warmup else adamw(ablation, lr=1e-3))
    losses = d2l.train_steps(ablation, data, optimizer, 800)
```

```
    print(f'pre_norm={pre_norm}, warmup={warmup}: loss at step '
          '100/200/400/800: ' + '/'.join(f'{sum(losses[k-50:k]) / 50:.2f}'
                                         for k in (100, 200, 400, 800)))
```

```
pre_norm=True, warmup=False: loss at step 100/200/400/800: 2.92/2.43/2.00/1.63
pre_norm=False, warmup=False: loss at step 100/200/400/800: 3.18/3.17/3.17/3.17
pre_norm=False, warmup=True: loss at step 100/200/400/800: 3.35/2.52/1.90/1.54
```

The pre-norm model trains normally. The post-norm model without warmup does not diverge, but its loss stops at about 3.2 nats and stays there. That is the unigram level computed above: the model predicts the byte frequencies of the text and ignores its input, even the current byte.

Attention that has learned nothing would not explain this level by itself. If every attention sublayer output zero, the feed-forward sublayers would still see the current byte and could reach the bigram level of 2.45 nats. The tiny query and key gradients at initialization are therefore at most part of the explanation. Xiong et al. (2020) showed that at initialization the gradients of the parameters near the output of a post-norm stack are large. A large learning rate without warmup then makes training unstable, and a warmup avoids the failure. The third run confirms this: with a warmup, the post-norm model leaves the unigram level and trains, while pre-norm needs no warmup at this rate. The warmup here is long, half the run, and an exercise asks how short it can be. GPT-2 already placed the normalization at the input of each sublayer and added one after the last block (Radford et al. 2019), the arrangement that `GPT` uses by default.

11.2.3 Sampling from the Model

A language model outputs a distribution over the next token, and text comes from decoding it. Section 8.7 treated greedy search, beam search, sampling, and the ways of reshaping the distribution before sampling. The sampling method below implements the two most common ways. The temperature $\tau > 0$ of Equation 8.32 divides the logits z_v before the softmax, $p_v \propto \exp(z_v/\tau)$. At $\tau = 1$ this is the model's own distribution. A smaller τ sharpens it, and as $\tau \to 0$ all probability goes to the most likely token, which is greedy decoding. A larger τ flattens it toward uniform. For the logits $(2, 1, 0)$, the probabilities are $(0.67, 0.24, 0.09)$ at $\tau = 1$, $(0.87, 0.12, 0.02)$ at $\tau = 0.5$, and $(0.51, 0.31, 0.19)$ at $\tau = 2$.

Top-k truncation (Fan et al. 2018) keeps only the k largest logits and sets the others to $-\infty$, which removes the long tail of unlikely tokens. The nucleus (top-p) and min-p rules of Section 8.7 truncate by probability instead and would fit into the same place of the loop.

We add a `generate` method to the class. It is deliberately simple. For every new token it runs the forward pass over the whole context again and recomputes the keys and values of every earlier position, although they have not changed. Section 11.3 measures that waste and removes it.

```
@d2l.add_to_class(GPT)
@torch.inference_mode()
def generate(self, prefix, num_tokens, temperature=1.0, top_k=None):
    """Sample a continuation of the token-id list prefix."""
    ids = torch.tensor(prefix, device=next(self.parameters()).device)
    for _ in range(num_tokens):  # The tokens stay on the device
```

```
            logits = self(ids[None, -self.max_len:])[0, -1] / temperature
            if top_k is not None:
                cutoff = torch.topk(logits, top_k).values[-1]
                logits[logits < cutoff] = -torch.inf
            ids = torch.cat((ids, torch.multinomial(F.softmax(logits, -1), 1)))
        return ids.tolist()
```

`generate` conditions on the last `max_len` tokens: once the text is longer than the model's context, the oldest tokens are cropped.

We sample 120 bytes after the prefix "The history of the" at three settings: the model's own distribution, a sharpened and truncated one, and a flattened one.

```
prefix = data.vocab.encode('The history of the ')
for temperature, top_k in ((1.0, None), (0.7, 8), (2.0, None)):
    out = model.generate(prefix, 120, temperature=temperature, top_k=top_k)
    d2l.print_wrapped(f'T={temperature}, top_k={top_k}: '
                      + repr(data.vocab.decode(out)))
```

```
T=1.0, top_k=None: 'The history of the Simpson Star Killerey quadron , with
    some semic hired as airframe . During the play Agnianines and Cup
    submitted Allied '
T=0.7, top_k=8: 'The history of the same set times to seat the country , and
    the building , including characters and the colonial class . \n The song
    is the '
T=2.0, top_k=None: 'The history of the bay\xe0rezerplTBalmitte (
    apherts\x05\xe0 \xe0\xe0 t\xe0xo \xe0\xe0\xe0\xe0\xe0\xe0\xe0\xe0\xe0\xe0 \xe0#.\xe0wLuxivvegelq\xe0pio9
    hhow\xe0 \xe0u\xe0`\xe0\xe0Ilbjo\xe0zly\u0268ndamo outh\xe0m orwa'
```

At $\tau = 1$ the model writes words and word-like strings in a plausible order, with the habits of its corpus, which sets punctuation off by spaces (" , " and " . "). The sharpened and truncated setting keeps to frequent words, most of them spelled correctly, and at $\tau = 2$ the bytes approach random strings. A loss near 1.35 nats per byte is a perplexity of about 3.9: at a typical position the model still hesitates among about four bytes, so the temperature changes the text visibly. Fluent text needs a much larger model trained on much more text, such as the one we load next.

11.2.4 Loading GPT-2

GPT-2 small is the `GPT` class with five settings (Radford et al. 2019): 12 blocks of width 768 with 12 heads, LayerNorm with biases, a GELU feed-forward network, learned positions for up to 1,024 tokens, and pre-norm with a tied head. Loading it takes a constructor call with these settings, the released weights, and the released tokenizer. We register the weight and tokenizer files in `d2l.DATA_HUB` with their content hashes, so that one download serves every later run. The tokenizer and the loader both go into the library, as `gpt2_tokenizer` and `load_gpt2`, because Section 11.3 studies how the released model uses its cache.

11.2.4.1 The Tokenizer

Our `GPT` reads bytes, but GPT-2 reads subword tokens, so loading it starts with its tokenizer. The cell registers the three released files and builds the tokenizer from the two that describe the vocabulary:

```
GPT2_URL = 'https://huggingface.co/openai-community/gpt2/resolve/main/'
d2l.DATA_HUB['gpt2-weights'] = (
    GPT2_URL + 'model.safetensors',
    '89a76996d7c6ee89b86618a265483aab73e61d50')
d2l.DATA_HUB['gpt2-merges'] = (
    GPT2_URL + 'merges.txt', '396d4d8ec90cb02f4d56e049e0e4add868bcd943')
d2l.DATA_HUB['gpt2-encoder'] = (
    GPT2_URL + 'vocab.json', 'f0223209235343bc067d7da838328bced8085ae1')

def gpt2_tokenizer():
    """GPT-2's byte-level BPE tokenizer, assembled by tiktoken."""
    import tiktoken  # lazy: d2l itself does not require tiktoken
    from tiktoken.load import data_gym_to_mergeable_bpe_ranks
    return tiktoken.Encoding(
        'gpt2', explicit_n_vocab=50257,
        pat_str=d2l.BPETokenizer.GPT2_PATTERN,
        mergeable_ranks=data_gym_to_mergeable_bpe_ranks(
            d2l.download('gpt2-merges', '../data/gpt2'),
            d2l.download('gpt2-encoder', '../data/gpt2')),
        special_tokens={'<|endoftext|>': 50256})

enc = gpt2_tokenizer()
ids = enc.encode('Attention is all you need.')
print(ids, [enc.decode([i]) for i in ids])
```

```
[8086, 1463, 318, 477, 345, 761, 13] ['Att', 'ention', ' is', ' all', ' you', ' need', '.']
```

The tokenizer applies the BPE machinery of Section 8.2 to GPT-2's released merge list and vocabulary. It uses the same pre-tokenization pattern as `d2l.BPETokenizer` and assembles the result into `tiktoken`'s encoder. The construction needs two data files and a regular expression, and no model library or configuration framework.

11.2.4.2 The Weight Map

The checkpoint stores one tensor per parameter, under the names of GPT-2's original code. The token table is `wte`, the position table `wpe`, and the final normalization `ln_f`. Block i is `h.i`. Inside it, `ln_1` and `ln_2` are the normalizations before attention and before the feed-forward network, `attn.c_attn` and `attn.c_proj` are our `W_qkv` and `W_o`, and `mlp.c_fc` and `mlp.c_proj` are our `W_1` and `W_2`. Loading therefore needs a dictionary of names and one rule about layout. GPT-2's code implemented its linear layers as a class `Conv1D` that stores each weight matrix with the input dimension first, as (in, out). The fused `c_attn` matrix concatenates queries, keys, and values in the same order as our `W_qkv`. The token embedding doubles as the output head in both models, and our LayerNorm uses GPT-2's $\epsilon = 10^{-5}$. Nothing else needs special handling. The function `load_gpt2` builds the configuration, copies every tensor into place, and returns the model in evaluation mode.

```
def load_gpt2():
    """GPT-2 small (124M) as a GPT with the released weights, in eval
    mode."""
    from safetensors.torch import load_file  # lazy, like tiktoken above
    gpt2 = GPT(vocab_size=50257, num_hiddens=768, num_heads=12, num_blks=12,
```

```
                max_len=1024, pos='learned', norm='layer', act='gelu',
                pre_norm=True, bias=True)
    weights = load_file(d2l.download('gpt2-weights', '../data/gpt2'))
    with torch.no_grad():
        gpt2.token_emb.weight.copy_(weights['wte.weight'])
        gpt2.pos_emb.weight.copy_(weights['wpe.weight'])
        gpt2.norm.weight.copy_(weights['ln_f.weight'])
        gpt2.norm.bias.copy_(weights['ln_f.bias'])
        for i, blk in enumerate(gpt2.blks):
            modules = {f'h.{i}.ln_1': blk.norm1, f'h.{i}.ln_2': blk.norm2,
                       f'h.{i}.attn.c_attn': blk.attention.W_qkv,
                       f'h.{i}.attn.c_proj': blk.attention.W_o,
                       f'h.{i}.mlp.c_fc': blk.ffn.W_1,
                       f'h.{i}.mlp.c_proj': blk.ffn.W_2}
            for key, module in modules.items():
                W = weights[key + '.weight']  # Conv1D: (in, out)
                module.weight.copy_(W.T if W.ndim == 2 else W)
                module.bias.copy_(weights[key + '.bias'])
    return gpt2.eval()

gpt2 = load_gpt2().to(d2l.try_gpu())
print(f'{d2l.num_params(gpt2) / 1e6:.1f}M parameters')
```

```
124.4M parameters
```

`nn.Linear` stores its weight as (out, in), the transpose of the `Conv1D` layout, so every 2-D weight is transposed on the way in. The copies run under `torch.no_grad`, because writing into a parameter in place is not an operation that autograd should record.

11.2.4.3 Checks of the Loaded Model

Shape checks alone do not validate a loaded checkpoint, because a transposed square matrix still multiplies. Two further checks test what the model computes. The first is quantitative: the model should assign a sentence of English a plausible probability.

```
text = ("The Time Machine, by H. G. Wells. The Time Traveller was "
        "expounding a recondite matter to us.")
x = torch.tensor(enc.encode(text), device=d2l.try_gpu())[None]
with torch.inference_mode():
    logits = gpt2(x)
loss = F.cross_entropy(logits[0, :-1], x[0, 1:])
print(f'per-token loss {loss.item():.2f}, '
      f'perplexity {loss.exp().item():.0f}')
```

```
per-token loss 3.89, perplexity 49
```

A perplexity around 50 means that the model is, on average, about as uncertain as a uniform choice among 50 tokens, where uniform guessing over the vocabulary would score 50,257. An exercise measures how much the common mapping mistakes raise it. The second check is qualitative: greedy decoding ($k = 1$) of a familiar prompt should continue it coherently.

```
out = gpt2.generate(enc.encode('Alan Turing theorized that computers '
                               'would one day become'), 16, top_k=1)
d2l.print_wrapped(enc.decode(out))
```

```
Alan Turing theorized that computers would one day become the most powerful
    machines on the planet.

The computer is a machine that
```

Sampling at $\tau = 0.8$ with $k = 50$ should give varied but grammatical text:

```
out = gpt2.generate(enc.encode('The secret of a good deep learning '
                               'textbook is'), 40, temperature=0.8,
                    top_k=50)
d2l.print_wrapped(enc.decode(out))
```

```
The secret of a good deep learning textbook is that its use takes time and
    effort. This means it's usually taken for granted that a well-written
    textbook is something you can learn over and over. But the problem with
    many deep learning textbooks is
```

The continuations are fluent English, unlike the samples of our own model. GPT-2 is the same class with about 26 times as many parameters, trained with tens of thousands of times as much computation on thousands of times as much text. The rest of this chapter keeps the class and changes what surrounds it: Section 11.3 makes generation fast, later sections add encoders, cross-attention, and mixtures of experts, and Section 11.7 asks how loss falls as parameters, data, and computation grow together.

11.2.5 Summary

A GPT predicts every next token of a window in one forward pass. A token embedding feeds a stack of transformer blocks whose attention is causally masked, and after a final normalization the same embedding table, used as the output head, produces the logits. Positions enter as a learned table added to the embedding, as in GPT-2, or as RoPE rotations inside each attention layer. GPT-2's initialization scales the projections that write into the residual stream by $1/\sqrt{2L}$, so that the stream does not grow with depth. We train with AdamW, weight decay on the matrices, warmup, cosine decay, and clipping, on a corpus so large that the model almost never reads the same passage twice, so its validation loss follows its training loss. Moving the normalization after the residual addition, at a learning rate that pre-norm tolerates without warmup, leaves the model at the unigram entropy, where its predictions ignore their context. A slow warmup lets the post-norm model train. Temperature and top-k truncation reshape the distribution from which text is sampled, and the released GPT-2 loads into the same class through a map of parameter names.

11.2.6 Exercises

1. **Parameter count.** The loading code of Section 11.2.4 reports 124.4M parameters for the GPT-2 configuration, which has width $d = 768$, 12 blocks, a vocabulary of $V = 50{,}257$ tokens, and 1,024 positions.

1. Count the parameters of one block exactly, including the biases of its four linear layers and the gains and shifts of its two LayerNorms. Add the token table, the position table, and the final normalization, and verify that the total is $124{,}439{,}808$.
2. Exercise 4 of Section 11.1 gives the exact parameter count of the default `TransformerBlock`, which `GPT` builds with its default flags. Use it to count the parameters of `GPT(vocab_size=256)`, with $d = 256$ and six blocks, and check the total against the count printed in Section 11.2.1. What fraction of the parameters sits in the embedding tables of each of the two models?
3. Without tying, the output head would be a separate $V \times d$ matrix. How many parameters would GPT-2 then have? A product of a $d_{\text{out}} \times d_{\text{in}}$ matrix with a vector costs about $2d_{\text{out}}d_{\text{in}}$ floating-point operations, and a table lookup costs none. Estimate the share of GPT-2's per-token matrix work that the head accounts for at short contexts, where the attention scores are negligible, and compare it with the share of the parameters that the shared table holds.

2. code **Unmasked attention.** In `GPT`, the flag `is_causal=True` of the attention call in `CausalAttention` is the only mechanism that keeps position t from attending to later positions.

   1. Take one validation window of 128 bytes and replace its byte at position 64 by a different one. For every position, compute the largest absolute change of the logits of the trained `model`, and plot the change against position.
   2. Subclass `GPT` with a nested `CausalAttention` whose attention call passes `is_causal=False`. Train it as in Section 11.2.2, with `LMOptimizer` set up for 1,000 steps, but stop after 200 steps. Compare its training and validation losses with those that the training of `model` recorded at step 200, and explain the numbers.
   3. Sample 100 bytes from the unmasked model with the prefix `'The history of the '`. Explain why the validation loss of the previous sub-problem and this sample judge the unmasked model so differently.

3. code **The recipe, one part at a time.** `LMOptimizer` combines four choices: the warmup, the cosine decay, the weight decay on the matrices, and the clipping.

   1. Train `GPT(len(data.vocab))` as in Section 11.2.2 five times: with the full recipe, and with each of the four choices removed in turn. Removing the decay of the learning rate means a constant rate after the warmup; removing the warmup means starting at the peak rate. Repeat each run three times and report the mean and the range of the final validation losses. Which removals change the loss by more than the run-to-run range?
   2. Repeat at three times the peak learning rate. Which of the four choices matter now, and why would you expect clipping and warmup to matter more at a high learning rate than at a low one?

4. code **How slow a warmup post-norm needs.** The third run of Section 11.2.2.5 warms the post-norm model up over 400 of its 800 steps.

   1. Repeat it with warmups of 100, 200, and 300 steps at the peak rate 10^{-3}, and with a warmup of 100 steps at 5×10^{-4}. Which runs leave the unigram level? Run each setting three times.
   2. Do your results depend on the length of the warmup, on the peak rate, or on the slope at which the rate rises? Design one more run that separates two of these explanations.
   3. Track the token spread of Section 11.1.2.2 at the last block during the first 100

steps of a run that stalls and of one that trains. Does the stalled run lose the distinction between its tokens, and if so, when?

5. code **Memorizing a novel.** Section 11.2.2 trains on WikiText-103 so that the model almost never reads the same passage twice. Here it reads one short novel many times.
   1. Train a `GPT` with the same recipe on `d2l.TimeMachine(batch_size=64, num_steps=128, tokenization='char', num_train=100000, num_val=3000)`, whose 100,000 overlapping training windows start at the first 100,000 positions of the text and whose 28 tokens come from the character pipeline of Section 8.2, so that the model has a table of 28 rows. Record both losses every 50 steps. At which step does the validation loss reach its minimum, and how far apart are the two losses at the end?
   2. A sample that copies its training text contains long strings that occur in that text verbatim. Draw ten samples of 100 tokens at temperature 1 without truncation, from this model with the prefix `'the time traveller '` and from `model` with the prefix `'The history of the '`. Compute the *verbatim fraction* of each sample, the fraction of its 20-token windows that occur verbatim in the corresponding training text.
   3. Retrain on *The Time Machine* with `dropout=0.3`. Does dropout move the minimum of the validation loss, lower it, or only slow the rise after it?

6. code **Checkpoint mismatches.** The GPT-2 checkpoint holds 160 tensors with 137,022,720 numbers in all, while `gpt2` has 124,439,808 parameters.
   1. Identify the checkpoint tensors that `load_gpt2` never reads, and explain why `GPT` needs no counterpart for them.
   2. Construct the GPT-2 configuration with `norm='rms'` and run the body of `load_gpt2` on it. Which line fails, and why can no mapping of the checkpoint onto this configuration reproduce GPT-2? Then list, for the GPT-2 configuration with `pos='rope'` and `act='swiglu'`, every checkpoint tensor without a destination and every model parameter without a source.
   3. Some mistakes load without any error. Introduce three of them, one at a time: skip every copy of a normalization shift and load the rest into the `norm='rms'` model; load the `attn.c_proj` matrix of every block in the wrong orientation, which no shape check detects because the matrix is square; and swap the query and key thirds of every `c_attn` weight and bias. For each mistake, report the perplexity of the sentence that Section 11.2.4 scores and the greedy continuation of the prompt `'Alan Turing theorized that computers would one day become'`. Which mistake does the perplexity check flag most clearly, and which least?
   4. A check that catches every such mistake compares the logits with an independent implementation. Write GPT-2's forward pass in NumPy directly from the checkpoint tensors, in float64 and with the `Conv1D` convention $\mathbf{xW} + \mathbf{b}$, and compare its logits for a short prompt with those of `gpt2`. How large is the largest difference relative to the largest logit, and what does that ratio say about where the difference comes from?

7. code **Tied head without blocks.** ⊠ Consider `GPT` with `num_blks=0` and its defaults `pos='rope'` and `norm='rms'`. Let $\mathbf{e}_1, \dots, \mathbf{e}_V \in \mathbb{R}^d$ be the rows of the token table, let γ be the gain of the final RMSNorm Equation 11.3, and let z_{ij} be the logit of token j when the current token is i.
   1. Show that $z_{ij} = s_i M_{ij}$ with scalars $s_i > 0$ and a symmetric matrix $\mathbf{M}$, and express s_i and $\mathbf{M}$ in terms of the table and γ.

2. For three distinct tokens i, j, and k, show that

$$\frac{z_{ij} - z_{ik}}{s_i} + \frac{z_{jk} - z_{ji}}{s_j} + \frac{z_{ki} - z_{kj}}{s_k} = 0.$$

   Conclude that on the periodic text $ijk\,ijk\,ijk\ldots$ the model assigns the correct next token a probability of at most $1/2$ in at least one of the three contexts, so that its average loss exceeds $(\log 2)/3$ nats per token. Which loss can a model with an untied head reach on this text?

3. Show that losses arbitrarily close to $(\log 2)/3$ are attainable.

4. Train `GPT(len(data.vocab), num_blks=0)` for 1,000 steps with `LMOptimizer` at a peak learning rate of 10^{-2}, and compare its final training loss with the bigram level of Section 11.2.2, using the first sub-problem to explain the difference.

8. code **Bits per byte against GPT-2.** GPT-2 and `model` use different tokens, so their per-token losses are not comparable. A loss of ℓ nats per token at c bytes per token corresponds to

$$\frac{\ell}{c \log 2} \text{ bits per byte.} \tag{11.9}$$

   1. Compute the bits per byte of `model` on the validation split of `data`.
   2. Compute the bits per byte of `gpt2` on the same text, in chunks of 1,024 tokens. WikiText keeps the tokenization of its creators: it sets punctuation off by spaces and writes a hyphen inside a word as `@-@`. Repeat on the text with these marks undone, still dividing by the bytes of the original text, as the GPT-2 paper did with invertible detokenizers.
   3. Which of the two GPT-2 figures should be compared with that of `model`? Discuss what each side brings to the passage: `model` was trained on text of the same source and format, sees 128 bytes of context, and predicts 256 symbols, whereas GPT-2 was trained on 40 GB of web text from which Wikipedia articles were removed, sees up to 1,024 tokens, and predicts 50,257 symbols.

9. [extended] **Subwords at small scale.** Section 11.2.2 trains on bytes, which keeps the embedding table small and makes the loss a number of bits per byte. Subword tokens have other advantages: a window of 128 tokens covers several times more text, and each prediction carries more information. Test which effect wins at this scale.

   1. Train a `d2l.BPETokenizer` with the GPT-2 pattern on the first million bytes of the training text for vocabulary sizes 1,024 and 4,096; with 256 it learns no merges and gives the bytes of `data`. Subclass `d2l.WikiText103` so that it encodes the first 20 million bytes of the training split and the whole validation split with a given tokenizer.
   2. Train `GPT` with each vocabulary as in Section 11.2.2 and convert the final validation losses to bits per byte as in 8. Count also the parameters in the embedding table of each model.
   3. A larger vocabulary gives each window more bytes of context and each step more text, so in 1,000 steps the model reads a larger share of its 20 million bytes. Repeat the best vocabulary with windows of about 128 bytes, and with the number of steps that reads as many bytes as the byte-level run. Which of the three effects explains the differences you found?

11.3 Generation and the KV Cache

The GPT of Section 11.2 writes text one token at a time, and its `generate` method produces each new token by running the model over the whole text so far. A chat assistant also writes one token at a time. If it were built like our GPT, it would answer a long question by reading the whole question again for every word of its answer. Does a model have to reprocess everything before it to choose the next token?

It does not. Attention is causal, so the keys and values of earlier tokens never change once computed. The *KV cache* stores them, and each step then runs the model on the new token alone. For a long prompt this removes all but a small fraction of the arithmetic.

This section builds the cache and measures the speedup. It then turns to the problem the cache creates. Generation becomes limited by memory rather than by arithmetic, because every generated token reads all weights of the model and the whole cache from memory. The cache of a long context, or of many users served together, outgrows the weights. We derive the size of the cache, measure how much faster a model reads a prompt than it writes text, and see why batching helps less as contexts grow. Then we shrink the cache in three ways: by sharing keys and values across heads, by compressing them to fewer dimensions, and by bounding the context with a sliding window that keeps a few early tokens. The experiments use `d2l.GPT` from Section 11.2, and the released GPT-2 weights for the effects that only a trained model shows.

11.3.1 From Recomputation to Caching

A forward pass of a model with N parameters over t tokens costs about $2Nt$ floating-point operations, because every parameter takes part in one multiplication and one addition per token. The attention scores add work that grows with t^2 (Section 10.5). At the lengths we run here it stays below $2Nt$, and we leave it out of the count, although at a few thousand tokens it is no longer small. Naive generation calls this forward pass once per new token, over a history that grows by one token per step. Producing T tokens therefore costs about $\sum_t 2Nt \approx NT^2$ operations, quadratic in the length of the text. Of the logits that each pass computes, only the last row is used.

Most of that work is repeated. At step t the model needs, in every layer, the attention output for one new query $\mathbf{q}_t$ against the keys $\mathbf{k}_1, \dots, \mathbf{k}_t$ and values $\mathbf{v}_1, \dots, \mathbf{v}_t$. The causal mask guarantees that positions $1, \dots, t-1$ never see position t, so their hidden states, and therefore their keys and values, are the same at step t as they were at step $t-1$. A KV cache stores, for every layer, the keys and values computed so far. Each step then runs the model on the one new token, writes its key and value into the cache, and attends to everything stored. Figure 11.4 contrasts the two schedules. A cached step costs about $2N$ operations, a forward pass over a single token, plus the scores of one query against t keys. For a model of GPT-2's size, with 124 million parameters, writing 256 tokens after a prompt of 3,584 tokens takes about 2.4×10^{14} operations without the cache. Reading the prompt once and then processing each new token once takes about 10^{12}, some 250 times less.

11.3.1.1 The Cached Forward Pass

Two details need care. The first is position. RoPE rotates queries and keys by their absolute positions, so a step must know the position t at which its new tokens start. `GPT.CausalAttention.qkv` takes this offset as an argument, and a model with learned positions looks up the embeddings of positions $t, t+1, \dots$ instead of $0, 1, \dots$. The RoPE angles depend

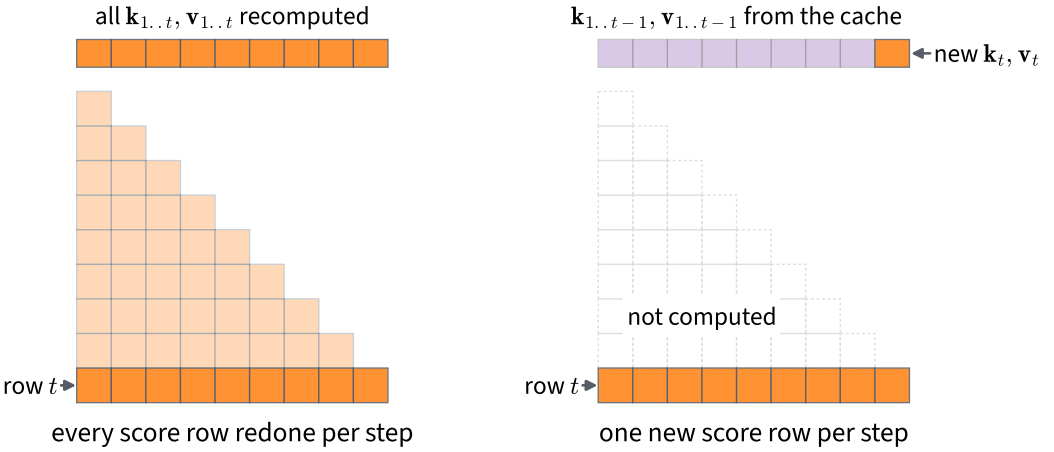


Figure 11.4: One decoding step at context length t. Naive generation recomputes every key, value, and score row of the prefix and then uses only the last row. With a KV cache, the past keys and values are read from memory, and the step computes a single new key, value, and score row.

on the positions alone, so a call computes their tables once with `d2l.rope_tables` and passes them to every block. The second is storage. Appending each new key to a growing tensor would copy the whole cache at every step, work that grows with the context. We therefore allocate the cache once, at its final size: for every block, one buffer of keys and one of values with room for `max_len` positions. A call with T new tokens writes their keys and values into positions $t, \dots, t+T-1$ and attends to the filled positions $0, \dots, t+T-1$.

The same function serves both phases of generation. In *prefill*, the whole prompt enters at $t = 0$, and its queries need the causal mask. In *decode*, one token at position t attends to all filled positions. The mask `d2l.causal_mask(T, t + T)` covers both cases, because it treats the T queries as the last T of $t+T$ positions (Section 10.2).

```
def init_cache(model, batch_size, max_len):
    """Zeroed keys and values for every block: two tensors of shape
    (blocks, batch, kv heads, max_len, head width)."""
    attn = model.blks[0].attention
    shape = (len(model.blks), batch_size, attn.num_kv_heads, max_len,
             attn.kv_hiddens // attn.num_kv_heads)
    p = next(model.parameters())
    return (torch.zeros(shape, device=p.device, dtype=p.dtype),
            torch.zeros(shape, device=p.device, dtype=p.dtype))

@torch.inference_mode()
def forward_cached(model, X, ks, vs, t):
    """Logits after the last of the new tokens X, which sit at positions
    t, t + 1, ...; writes their keys and values into the cache."""
    assert all(blk.pre_norm for blk in model.blks), 'pre-norm blocks only'
    T = X.shape[1]
    H, tables = model.token_emb(X), None
```

```
    if model.pos == 'learned':
        H = H + model.pos_emb(torch.arange(t, t + T, device=X.device))
    else:  # RoPE angles of positions t, ..., t + T - 1, for every block
        tables = d2l.rope_tables(T, model.head_width, t, device=X.device)
    mask = d2l.causal_mask(T, t + T, X.device)
    for i, blk in enumerate(model.blks):  # Pre-norm, as in d2l.GPT
        attn = blk.attention
        q, k, v = attn.qkv(blk.norm1(H), t, tables)  # Rotated at t, ...
        ks[i, :, :, t:t + T], vs[i, :, :, t:t + T] = k, v
        Y = F.scaled_dot_product_attention(
            q, ks[i, :, :, :t + T], vs[i, :, :, :t + T], attn_mask=mask,
            enable_gqa=attn.num_kv_heads < attn.num_heads)
        H = H + attn.W_o(Y.transpose(1, 2).flatten(2))
        H = H + blk.ffn(blk.norm2(H))
    return F.linear(model.norm(H[:, -1]), model.token_emb.weight), ks, vs
```

The buffers are written in place, so `forward_cached` returns the same `ks` and `vs` that it received; returning them keeps one calling convention for the functions below. The slice `ks[i, :, :, :t + T]` is a view into the buffer, not a copy. Computing the RoPE tables once per call instead of in every block saves about a quarter of the time of a decode step, which, as we will see, goes mostly to launching small kernels.

Generation is a prefill call over the prompt followed by one decode call per new token. `decode` draws each token from the last logits, after the temperature and top-k truncation of Section 11.2.3, and feeds it back. `generate_cached` gives it the interface of `generate`.

```
def sample(logits, temperature=1.0, top_k=None):
    """One token per row of logits (batch, vocab), as generate draws it."""
    logits = logits / temperature
    if top_k is not None:
        cutoff = torch.topk(logits, top_k).values[:, -1:]
        logits = logits.masked_fill(logits < cutoff, -torch.inf)
    return torch.multinomial(F.softmax(logits, -1), 1)

@torch.inference_mode()
def decode(model, X, ks, vs, t, num_tokens, temperature=1.0, top_k=None):
    """Feed X at positions t, t + 1, ..., then sample num_tokens tokens
    per sequence; returns them as (batch, num_tokens), and the cache."""
    logits, ks, vs = forward_cached(model, X, ks, vs, t)
    t, tokens = t + X.shape[1], []
    for i in range(num_tokens):
        tokens.append(sample(logits, temperature, top_k))
        logits, ks, vs = forward_cached(model, tokens[-1], ks, vs, t + i)
    return torch.cat(tokens, 1), ks, vs

@d2l.add_to_class(d2l.GPT)
def generate_cached(self, prefix, num_tokens, temperature=1.0, top_k=None):
    """Sample as generate does, but never recompute the prefix."""
    assert len(prefix) + num_tokens <= self.max_len, 'cache overflow'
    X = torch.tensor(prefix, device=next(self.parameters()).device)[None]
    ks, vs = init_cache(self, 1, len(prefix) + num_tokens)
    tokens = decode(self, X, ks, vs, 0, num_tokens, temperature, top_k)[0]
```

```
    return list(prefix) + tokens[0].tolist()
```

11.3.1.2 Checking the Cache

The cached path must reproduce the logits of the full forward pass. For both positional schemes, we prefill the first 16 tokens of a random sequence, decode the remaining 32 one at a time, and compare the logits after each position with those of one forward pass over the whole sequence. We build every model of this section through one shorthand:

```
gpt = d2l.GPT
```

```
for pos in ('rope', 'learned'):
    model = gpt(vocab_size=97, num_hiddens=128, num_heads=4, num_blks=3,
                pos=pos).eval()
    x = torch.randint(0, 97, (1, 48))
    with torch.inference_mode():
        full = model(x)[0, 15:]  # Logits after positions 15, ..., 47
    ks, vs = init_cache(model, 1, 48)
    logits, ks, vs = forward_cached(model, x[:, :16], ks, vs, 0)
    outs = [logits]
    for t in range(16, 48):
        logits, ks, vs = forward_cached(model, x[:, t:t + 1], ks, vs, t)
        outs.append(logits)
    err = (torch.cat(outs) - full).abs().max()
    print(f'{pos}: max |full - cached| = {err:.1e}')
```

```
rope: max |full - cached| = 1.2e-06
learned: max |full - cached| = 6.0e-07
```

The two paths agree to rounding error. Next we time one generation step at growing context lengths, on an untrained model of GPT-2's size: 12 blocks of width 768, 124 million parameters. The arithmetic does not depend on the weights.

```
device = d2l.try_gpu()
model = gpt(vocab_size=50257, num_hiddens=768, num_heads=12, num_blks=12,
            max_len=4096).to(device).eval()  # GPT-2's vocabulary and size
print(f'{d2l.num_params(model) / 1e6:.0f}M parameters')
```

```
124M parameters
```

Two helpers time the two kinds of step. The naive step, `time_forward`, is a forward pass over all n tokens. The cached step, `time_decode`, is a decode call against a cache of n positions, averaged over 32 steps, for a batch of sequences; we reuse it below. Both time with `d2l.Benchmark`, which calls a function a few times to warm up, waits for the device to finish, and averages the time of ten further calls.

```
def time_forward(model, n):
    """Seconds for a forward pass over n random tokens."""
    device = next(model.parameters()).device
```

```
    x = torch.randint(0, model.token_emb.num_embeddings, (1, n),
                      device=device)
    with torch.inference_mode():
        return d2l.Benchmark(lambda: model(x)[:, -1]).time

def time_decode(model, batch_size, n):
    """Seconds per decode step for batch_size sequences at context n,
    averaged over 32 steps."""
    device = next(model.parameters()).device
    x = torch.randint(0, model.token_emb.num_embeddings, (batch_size, 1),
                      device=device)
    ks, vs = init_cache(model, batch_size, n + 32)
    return d2l.Benchmark(
        lambda: decode(model, x, ks, vs, n - 1, 31)).time / 32
```

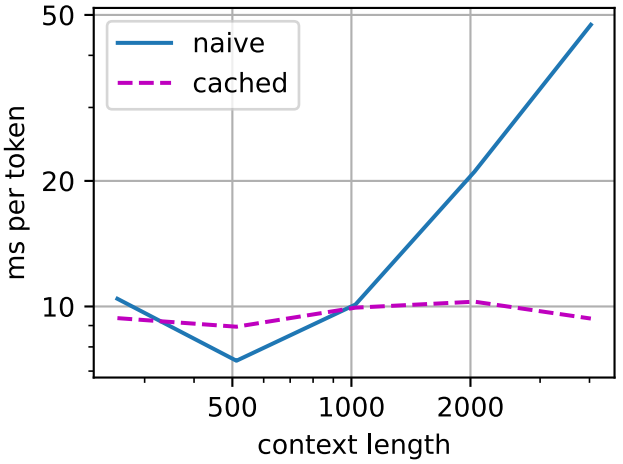
We time both steps at five context lengths:

```
lengths, t_naive, t_cached = [256, 512, 1024, 2048, 4032], [], []
for n in lengths:
    t_naive.append(time_forward(model, n))
    t_cached.append(time_decode(model, 1, n))
    print(f'n={n:4d}: naive {1e3 * t_naive[-1]:5.1f} ms, '
          f'cached {1e3 * t_cached[-1]:4.1f} ms per token')
```

```
n= 256: naive  10.4 ms, cached  9.4 ms per token
n= 512: naive   7.4 ms, cached  8.9 ms per token
n=1024: naive  10.1 ms, cached  9.9 ms per token
n=2048: naive  21.0 ms, cached 10.3 ms per token
n=4032: naive  47.4 ms, cached  9.4 ms per token
```

On logarithmic axes, a time proportional to n is a line of slope one.

```
d2l.plot(lengths, [[1e3 * t for t in t_naive], [1e3 * t for t in t_cached]],
         'context length', 'ms per token', legend=['naive', 'cached'],
         xscale='log', yscale='log')
```



At long contexts the naive step grows at least in proportion to n, as $2Nn$ says it must, and faster where the attention scores, whose work grows with n^2, add a sizable share. The cached step grows far more slowly. The time to read the cache does grow with n, but even at 4,000 tokens the cache is smaller than the weights, and every step also has a fixed cost that does not depend on n. At the longest context the cached step is more than five times faster, and beyond about a thousand tokens the gap widens with every doubling of the context. At short contexts the two curves approach each other and can even cross. A step of either kind then spends much of its time on fixed costs, such as launching the kernels of twelve blocks, rather than on arithmetic.

End to end, we generate 256 tokens after a prompt of 3,584 tokens with both methods, after one warm-up call each:

```
prefix = list(range(3584))  # Any token ids will do for timing
for name, gen in (('naive ', model.generate),
                  ('cached', model.generate_cached)):
    t = d2l.Benchmark(lambda: gen(prefix, 256), warmup=1, repeats=1).time
    print(f'{name}: 256 tokens in {t:4.1f} s = {256 / t:5.1f} tokens/s')
```

```
naive : 256 tokens in 10.9 s =  23.4 tokens/s
cached: 256 tokens in  2.5 s = 103.1 tokens/s
```

The cached loop is more than five times faster, though by far less than the factor of about 250 in operations. Every decoded token still reads all the weights of the model, as Section 11.3.2.1 shows. Serving systems report the two phases separately: the *time to first token* measures the prefill, and the *tokens per second* measure the decode loop.

11.3.2 Memory and Bandwidth

The cache removes the repeated arithmetic, but it has to be stored, and it is read again at every step, so we start with its size. The size of the cache follows from the shape of its buffers. Each of the L blocks stores one key and one value vector of dimension $H_{\text{kv}}d_h$ per token, for H_{kv} key–value heads of width d_h, so a batch of b sequences of length n holds

$$\text{cache bytes} \;=\; 2 \cdot L \cdot H_{\text{kv}} \cdot d_h \cdot n \cdot b \cdot (\text{bytes per element}), \tag{11.10}$$

where the leading 2 counts keys and values. So far H_{kv} equals the number H of query heads; Section 11.3.3.2 breaks that equality. For our GPT-2-sized model in fp32 the cache takes $2 \cdot 12 \cdot 12 \cdot 64 \cdot 4$ bytes $= 72$ KiB per token. At a context of 4,096 tokens that is 288 MiB, more than half the size of the model's weights. A deployed model has dozens of layers, contexts of hundreds of thousands of tokens, and many users served at once. Then the cache, not the weights, fills the accelerator; an exercise works this out for a 70B model. We save the formula for any `d2l.GPT`.

```
def kv_cache_bytes(model, num_tokens, batch_size=1, bytes_per_elem=4):
    """Bytes of the keys and values of num_tokens positions in every block
    of a GPT, for batch_size sequences."""
    attn = model.blks[0].attention
    return (2 * len(model.blks) * attn.kv_hiddens * num_tokens * batch_size
            * bytes_per_elem)
```

The buffers of `init_cache` have this size, so a cache that does not fit into memory fails when it is allocated, before generation starts.

```
bytes_per_param = 4  # fp32 weights
weight_bytes = bytes_per_param * d2l.num_params(model)
ks, vs = init_cache(model, 1, 4096)
print(f'cache buffers {(ks.nbytes + vs.nbytes) / 2**20:.0f} MiB, '
      f'formula {kv_cache_bytes(model, 4096) / 2**20:.0f} MiB, '
      f'weights {weight_bytes / 2**20:.0f} MiB')
```

```
cache buffers 288 MiB, formula 288 MiB, weights 471 MiB
```

11.3.2.1 Prefill and Decode

At each generated token the whole cache is read from memory, and so is every weight of the model, because a forward pass over one token multiplies its activations by every weight matrix. Which resource limits a computation, arithmetic or memory traffic, depends on its *arithmetic intensity*: the number of floating-point operations per byte moved between memory and the processor. In the *roofline* model of performance, which Chapter 13 develops, a computation takes at least its operations divided by the peak rate of the processor. It also takes at least its bytes divided by the memory bandwidth, and the larger of the two bounds sets its time. An RTX 4090 performs up to about 83 TFLOP/s in fp32 but moves only about 1 TB/s, so a computation needs about 80 FLOPs per byte to keep the arithmetic units busy.

A decode step performs about $2N$ operations and reads at least the $4N$ bytes of the fp32 weights plus the cache, an intensity below one FLOP per byte. A prefill over n tokens performs $2Nn$ operations with the same weight traffic, because each weight is read once and used for every token, so its intensity is about n times higher. Prefill is therefore limited by arithmetic and decode by memory bandwidth (Pope et al. 2023). We time a forward pass over a 2,048-token prompt and the decoding of 32 tokens after it.

```
t_prefill = time_forward(model, 2048)
t_decode = time_decode(model, 1, 2048)
```

We compare both rates with the intensity of a decode step and with the rate that a bandwidth of 1 TB/s would allow.

```
bandwidth = 1e12  # bytes/s, about an RTX 4090
flops = 2 * d2l.num_params(model)  # One decode step at context 2048
moved = weight_bytes + kv_cache_bytes(model, 2048)
print(f'prefill: {2048 / t_prefill:6.0f} tokens/s')
print(f'decode:  {1 / t_decode:6.0f} tokens/s, bandwidth ceiling '
      f'{bandwidth / moved:.0f} tokens/s')
print(f'FLOPs per byte: decode {flops / moved:.2f}, '
      f'prefill about {2048 * flops / moved:.0f}')
```

```
prefill:  97076 tokens/s
decode:      98 tokens/s, bandwidth ceiling 1550 tokens/s
FLOPs per byte: decode 0.38, prefill about 784
```

The model reads the prompt at about a hundred thousand tokens per second and generates tokens at least an order of magnitude more slowly, with the same layers on the same hardware. The decode rate also stays below the bandwidth ceiling, because a fixed cost per step, such as launching the kernels of twelve blocks, adds to the time to read the weights. The bytes moved per token, weights and cache together, still set the limit that no implementation can beat. A smaller cache therefore allows faster decoding, longer contexts, or more sequences at once.

11.3.2.2 Batched Decoding

A decode step for one sequence reads all W bytes of the weights to produce one token. A step for a batch of b sequences reads the weights once and produces b tokens, because every sequence multiplies its activations by the same matrices. The cache gains nothing from batching. Each sequence reads its own nc bytes, where c is the cache size per token from Equation 11.10. A step therefore moves about $W + bnc$ bytes. When the memory bandwidth β limits the step, the throughput is

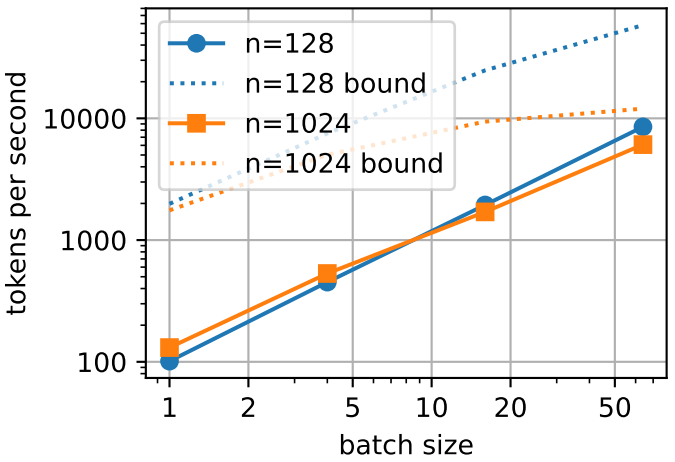
$$\text{tokens per second} \approx \frac{b\beta}{W + bnc}. \tag{11.11}$$

For small b the throughput grows in proportion to b. Once the caches outweigh the weights, at $b > W/(nc)$, it levels off toward $\beta/(nc)$, a ceiling set by the cache alone. For our fp32 model, $W \approx 0.5$ GB and $c = 72$ KiB. The caches equal the weights at a batch of about 53 for a context of 128 tokens, but already at a batch of about 7 for 1,024 tokens. We measure the decode throughput at both contexts for batches of 1 to 64 sequences and plot it next to the bound of Equation 11.11 with $\beta = 1$ TB/s.

```
batches, curves = [1, 4, 16, 64], []
for n in (128, 1024):
    rates = [b / time_decode(model, b, n) for b in batches]
```

```
    nc = kv_cache_bytes(model, n)  # One sequence's cache
    curves += [rates,
               [b * bandwidth / (weight_bytes + b * nc) for b in batches]]
    print(f'n={n}: ' + ', '.join(f'{r:.0f}' for r in rates) + ' tokens/s')
d2l.plot(batches, curves, 'batch size', 'tokens per second',
         legend=['n=128', 'n=128 bound', 'n=1024', 'n=1024 bound'],
         fmts=('C0-o', 'C0:', 'C1-s', 'C1:'), xscale='log', yscale='log')
```

```
n=128: 101, 452, 1943, 8524 tokens/s
n=1024: 131, 532, 1704, 6085 tokens/s
```



The bound keeps growing over the whole range of batches at a context of 128 tokens, since the caches reach the size of the weights only near a batch of 53. At 1,024 tokens it grows little beyond a batch of 16 and approaches the ceiling set by the caches. Batching spreads the weight reads over many sequences, but every sequence still reads its own cache. At small batches the measured curves stay below the bound, by the fixed cost per step of the previous subsection.

Here the fixed cost dominates almost the whole range. Each decode step launches several hundred small kernels from Python, one or more per operation of every block, and launching them takes longer than the GPU needs to run them, weight and cache reads included. The time per step therefore hardly depends on the batch or the context: the measured throughput grows in proportion to the batch, far below the bound. Only at 64 sequences of 1,024 tokens does the GPU's work, most of it reading the caches, catch up with the launches. Section 13.3 removes this cost by recording the kernels of a step once and replaying them (CUDA graphs) or by compiling the step; only then does the bound of Equation 11.11 appear in the measurement.

When many users are served, the batch changes from step to step, because requests arrive and finish at different times and with contexts of different lengths. Serving systems therefore admit and retire sequences at every decode step, which is called *continuous batching* (Yu et al. 2022). They also store the cache in fixed-size blocks located through a table, like the pages of virtual memory, instead of in one buffer of maximal length per sequence (Kwon et al. 2023). Section 39.7 shows these engines in use, together with speculative decoding, and the Language Models part of the book builds their mechanisms.

11.3.3 Sharing Keys and Values across Heads

In Equation 11.10, the factor H_{kv} counts key–value heads, and nothing in the attention mechanism forces it to equal the number H of query heads. *Multi-query attention* (MQA) keeps all H query heads but a single key–value head shared by all of them (Shazeer 2019), which shrinks the cache by a factor of H at a usually slight cost in quality. *Grouped-query attention* (GQA) lies in between, with H_{kv} key–value heads, each serving a group of H/H_{kv} query heads (Ainslie et al. 2023). Figure 11.5 shows the sharing patterns. GQA with H_{kv} between $H/8$ and $H/4$ is the common choice in current open models: Llama 3 8B and Mistral 7B both run 32 query heads against 8 key–value heads (Grattafiori et al. 2024; Jiang et al. 2023).

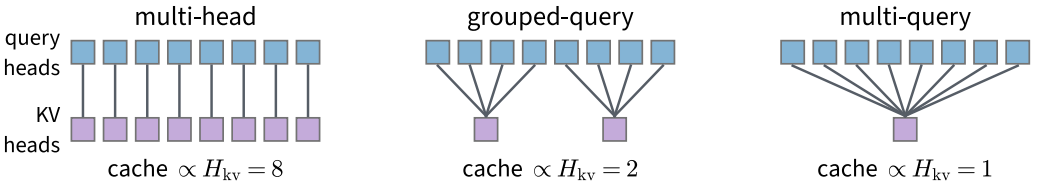


Figure 11.5: Query heads and the key–value heads they read. Multi-head attention gives every query head its own key–value head; grouped-query attention shares one per group of query heads; multi-query attention shares a single one across all of them. The cache scales with the number of key–value heads.

11.3.3.1 Grouped Key–Value Heads in Code

`d2l.GPT` and its attention `GPT.CausalAttention` take the number of key–value heads as the argument `num_kv_heads` (Section 11.2). The fused projection $\mathbf{W}_{qkv}$ then produces H query heads but only H_{kv} key heads and H_{kv} value heads. Query head h reads key–value head $\lfloor hH_{\text{kv}}/H \rfloor$, so each group of H/H_{kv} consecutive query heads shares one, and the cache stores each key–value head once. With $H_{\text{kv}} = H$ the layer is ordinary multi-head attention, and the cached forward pass above handles any H_{kv}, since `init_cache` sizes the buffers by `num_kv_heads`.

```
attn = d2l.GPT.CausalAttention(256, num_heads=8, num_kv_heads=2)
q, k, v = attn.qkv(torch.ones(2, 10, 256))
print(f'queries {tuple(q.shape)}, keys {tuple(k.shape)}, '
      f'values {tuple(v.shape)}')
```

```
queries (2, 8, 10, 32), keys (2, 2, 10, 32), values (2, 2, 10, 32)
```

Eight query heads of width 32 face two key heads and two value heads. The parameter count shows where the saving comes from: $\mathbf{W}_k$ and $\mathbf{W}_v$ shrink with H_{kv}, so the attention parameters fall from $4d^2$ toward $2d^2$, and the cache per token shrinks by the factor H/H_{kv}. We count both in a `d2l.GPT` of a single block.

```
for G in (8, 4, 2, 1):
    lm = gpt(256, num_hiddens=256, num_heads=8, num_blks=1, num_kv_heads=G)
    print(f'H_kv={G}: {d2l.num_params(lm.blks[0].attention):6d} '
          f'parameters, cache {kv_cache_bytes(lm, 1):5d} bytes per token '
          'and block')
```

```
H_kv=8: 262144 parameters, cache  2048 bytes per token and block
H_kv=4: 196608 parameters, cache  1024 bytes per token and block
H_kv=2: 163840 parameters, cache   512 bytes per token and block
H_kv=1: 147456 parameters, cache   256 bytes per token and block
```

11.3.3.2 Cache Size and Validation Loss

Sharing helps only if the model stays nearly as good. To measure its effect, we train the byte-level model of Section 11.2.2 with its recipe, once for each number of key–value heads from $H_{\text{kv}} = 8$ (full multi-head attention) down to $H_{\text{kv}} = 1$ (MQA). The data are the bytes of the Wikipedia articles of `d2l.WikiText103`, and `d2l.LMOptimizer` runs AdamW with warmup, cosine decay, and gradient clipping for 1,000 steps. The model almost never reads the same passage twice, so the validation loss at the end of training measures what it has learned.

```
data = d2l.WikiText103(batch_size=64, num_steps=128)
for G in (8, 4, 2, 1):
    lm = gpt(len(data.vocab), num_kv_heads=G)
    optimizer = d2l.LMOptimizer(lm, lr=2e-3, num_steps=1000, warmup=100)
    d2l.train_steps(lm, data, optimizer, 1000)
    loss = math.log(d2l.lm_perplexity(lm, data.val_dataloader()))
    print(f'H_kv={G}: validation loss {loss:.2f}, cache '
          f'{kv_cache_bytes(lm, 1) / 1024:4.1f} KiB per token')
```

```
H_kv=8: validation loss 1.34, cache 12.0 KiB per token
H_kv=4: validation loss 1.34, cache  6.0 KiB per token
H_kv=2: validation loss 1.34, cache  3.0 KiB per token
H_kv=1: validation loss 1.35, cache  1.5 KiB per token
```

The cache shrinks eightfold from the first row to the last. All four losses lie within a few hundredths of a nat, so at this scale even a single key–value head causes no loss we can measure. At large scale MQA loses a little quality, while GQA with $H/8$ key–value heads comes close to multi-head attention (Ainslie et al. 2023). A multi-head checkpoint can also be *uptrained* into a grouped one. Averaging the key and value projections within each group and training on for about 5% of the original steps recovers most of the quality, and this is how GQA spread through existing model families.

11.3.4 Compressing the Cache Further

Sharing heads reduces one factor of Equation 11.10. Two other factors can shrink as well: the width of what is stored per token, and the number of tokens stored. Our small byte-level models are too weak to show how a trained model uses its cache, so we study both factors on the released GPT-2. `d2l.load_gpt2` builds it as a `d2l.GPT` with the weights loaded as in Section 11.2.4, and `d2l.gpt2_tokenizer` returns its tokenizer. We encode *The Time Machine* and keep its first 1,024 tokens as the evaluation passage `x` and the next 1,024 tokens as a second passage `calib`.

```
gpt2, enc = d2l.load_gpt2().to(device), d2l.gpt2_tokenizer()
fname = d2l.download(d2l.DATA_URL + 'timemachine.txt', '../data',
                     '090b5e7e70c295757f55df93cb0a180b9691891a')
tokens = torch.tensor(enc.encode(open(fname).read()), device=device)
x, calib = tokens[None, :1024], tokens[None, 1024:2048]
```

```
print(f'{len(tokens)} tokens of The Time Machine')
```

```
44073 tokens of The Time Machine
```

Each experiment below changes only what happens inside attention, so one helper serves them all: a forward pass that walks GPT-2's blocks as the class does, but computes attention with a function that we supply. We check that the helper with ordinary causal attention reproduces the model, and compute a baseline loss on the second half of the passage, from position 512 on, where every predicted token has a long history.

```
def forward_blocks(model, X, attn_fn):
    """The forward pass of a d2l.GPT with attention computed by
    attn_fn(q, k, v, i) in block i, on inputs of shape (batch, heads,
    length, head width)."""
    assert all(blk.pre_norm for blk in model.blks), 'pre-norm blocks only'
    H = model.token_emb(X)
    if model.pos == 'learned':
        H = H + model.pos_emb(torch.arange(X.shape[1], device=X.device))
    for i, blk in enumerate(model.blks):
        attn = blk.attention
        Y = attn_fn(*attn.qkv(blk.norm1(H)), i)
        H = H + attn.W_o(Y.transpose(1, 2).flatten(2))
        H = H + blk.ffn(blk.norm2(H))
    return F.linear(model.norm(H), model.token_emb.weight)

def tail_loss(logits, X):
    """Mean loss of the next-token predictions from position 512 on."""
    return F.cross_entropy(logits[0, 512:-1], X[0, 513:])

causal = lambda q, k, v, i: F.scaled_dot_product_attention(
    q, k, v, is_causal=True)
with torch.inference_mode():
    logits = forward_blocks(gpt2, x, causal)
    print(f'helper vs model: max deviation '
          f'{(logits - gpt2(x)).abs().max():.1e}')
    full = tail_loss(logits, x)
print(f'full attention: loss {full:.2f}, perplexity {full.exp():.0f}')
```

```
helper vs model: max deviation 0.0e+00
full attention: loss 3.58, perplexity 36
```

11.3.4.1 Low-Rank Keys and Values

GQA shrinks the cache by storing fewer key–value heads. A different approach keeps all heads. It assumes that the keys and values of a token, $2 \times 768 = 1536$ numbers in GPT-2, lie close to a subspace of much lower dimension, so that the cache can store the r coordinates of each token in that subspace. *Multi-head latent attention* (MLA), around which DeepSeek built its V2 and V3 models, implements this idea (DeepSeek-AI 2024). A trained down-projection compresses the keys and values of each token jointly into one latent vector, and the cache stores only the latent. In DeepSeek-V2 the latent has 512 dimensions, against 32,768 numbers of keys and values per token in its 128 heads. Trained up-projections reconstruct the keys and values of every head when they are read. RoPE needs one more piece.

Its rotation depends on the position, so it cannot be folded into a shared projection, and MLA carries a small separate rotary key per token.

Whether the keys and values of a trained model admit such a compression can be tested on GPT-2 without training anything. In every block we stack the keys and values of each token of `calib` as one row of a matrix, and compute its top r right singular vectors. By the Eckart–Young theorem they span the r-dimensional subspace closest to these rows (Section 33.5). We then run the model on `x` with every key and value replaced by its projection onto this subspace, which a cache could store as r coordinates per token. The basis comes from another passage, as a trained projection would, so no token of `x` influences how the keys and values of `x` are compressed.

```
def kv_rows(k, v):
    """Keys and values of all heads of each token as one row: (T, 2d)."""
    return torch.cat([u[0].transpose(0, 1).flatten(1) for u in (k, v)], 1)

def fit_bases(model, X):
    """Right singular vectors of the key-value rows of X in every block."""
    bases = []
    def fit(q, k, v, i):
        bases.append(torch.linalg.svd(kv_rows(k, v), full_matrices=False)[2])
        return causal(q, k, v, i)
    forward_blocks(model, X, fit)
    return bases

def projected(bases, r):
    """Attention with keys and values projected onto the top r bases."""
    def attn_fn(q, k, v, i):
        P = bases[i][:r]  # (r, 2d), orthonormal rows
        KV = kv_rows(k, v) @ P.T @ P
        k, v = (u.unflatten(1, (k.shape[1], -1)).transpose(0, 1)[None]
                for u in KV.chunk(2, 1))
        return causal(q, k, v, i)
    return attn_fn

with torch.inference_mode():
    bases = fit_bases(gpt2, calib)
    for r in (512, 256, 128):
        loss = tail_loss(forward_blocks(gpt2, x, projected(bases, r)), x)
        print(f'rank {r} of 1536: loss {loss:.2f}, '
              f'perplexity {loss.exp():.0f}')
```

```
rank 512 of 1536: loss 3.55, perplexity 35
rank 256 of 1536: loss 3.61, perplexity 37
rank 128 of 1536: loss 4.14, perplexity 63
```

With a basis of rank 512, a third of the width, the loss matches that of full attention, even a few hundredths of a nat below it, since dropping the weakest directions also drops some noise. At rank 256, a sixth, it rises by only a few hundredths of a nat, although the basis never saw the passage. At rank 128 the loss rises by about half a nat. The keys and values of a trained model thus lie close to a subspace much smaller than their width, and the subspace carries over from one passage of the book to the next. MLA builds on this structure but learns its projections during training instead of fitting them afterward, so the model can

adapt to the compression.

11.3.4.2 Sliding Windows and Attention Sinks

The remaining factor is length. Keeping only the most recent w tokens in the cache, and evicting the oldest as generation proceeds, bounds its memory for any length of text. This is the rolling-buffer form of the sliding-window attention of Section 10.5. Before evicting anything, we measure where a trained model sends its attention: the average weight that GPT-2's queries, from position 64 on, place on the first token of the passage.

```
def sink_mass(model, X):
    """Mean attention weight on token 0 of the queries from position 64
    on, one entry per block."""
    masses = []
    def watch(q, k, v, i):  # Explicit causal attention
        scores = q @ k.transpose(-1, -2) / math.sqrt(q.shape[-1])
        alpha = d2l.masked_softmax(scores, d2l.causal_mask(
            q.shape[2], device=q.device))
        masses.append(alpha[..., 64:, 0].mean())
        return alpha @ v
    forward_blocks(model, X, watch)
    return masses

with torch.inference_mode():
    masses = sink_mass(gpt2, x)
print('mean attention weight on token 0 (uniform would be about 0.003)')
print('block ' + ''.join(f'{i:6d}' for i in range(len(masses))))
print('weight' + ''.join(f'{m:6.2f}' for m in torch.stack(masses).tolist()))
```

```
mean attention weight on token 0 (uniform would be about 0.003)
block      0     1     2     3     4     5     6     7     8     9    10    11
weight  0.00  0.01  0.11  0.30  0.32  0.51  0.46  0.57  0.48  0.48  0.45  0.34
```

From block 3 on, the queries place between about a third and more than half of their attention on the first token, a hundred times its uniform share or more. Xiao et al. (2024) call such positions *attention sinks* and trace them to the softmax. The weights of every query sum to one, so a head with nothing to retrieve must still put its weight somewhere. The first token, which every query can see, becomes that place.

A rolling buffer of the most recent w tokens evicts this token as soon as the text is longer than the window, and the softmax then spreads its weight over tokens that the heads had chosen to ignore. StreamingLLM therefore keeps the first few tokens in the cache next to the window (Xiao et al. 2024). We reproduce both the failure and the fix by evaluating the passage with attention restricted to a window of 256 tokens plus a number of retained *sink tokens* at the start.

```
j = torch.arange(x.shape[1], device=device)
allowed = d2l.causal_mask(x.shape[1], device=device)
w = 256  # The window: each query sees itself and the 255 tokens before it
window = allowed & (j[:, None] - j[None, :] < w)
with torch.inference_mode():
    for nsink in (0, 1, 4):
```

```
        mask = window | (allowed & (j[None, :] < nsink))
        loss = tail_loss(forward_blocks(
            gpt2, x, lambda q, k, v, i: F.scaled_dot_product_attention(
                q, k, v, attn_mask=mask)), x)
        print(f'window {w} + {nsink} sink tokens: loss {loss:.2f}, '
              f'perplexity {loss.exp():.0f}')
print(f'full attention:            loss {full:.2f}, '
      f'perplexity {full.exp():.0f}')
```

```
window 256 + 0 sink tokens: loss 8.25, perplexity 3836
window 256 + 1 sink tokens: loss 3.86, perplexity 47
window 256 + 4 sink tokens: loss 3.76, perplexity 43
full attention:            loss 3.58, perplexity 36
```

Without sink tokens, the window raises the perplexity from 36 to several thousand. Keeping the first token removes almost all of that increase, and with four tokens the loss comes within about 0.2 nat of full attention. The rest of the gap is what the window loses by dropping older tokens. Our mask keeps the original position of every token. A real rolling buffer must also decide which positions the retained entries carry, and an exercise works this out.

Models can also be trained not to need a sink. gpt-oss gives every head of every attention layer a learned *sink logit*, an extra entry in the denominator of its softmax that absorbs unneeded weight, so the first token no longer has to serve this role (OpenAI 2025). Qiu et al. (2025) multiply the output of each head by a sigmoid gate computed from the layer's input, so a head with nothing to retrieve can return nothing. In their models with 1.7 and 15 billion parameters, the average attention on the first token fell from 47% to 5%.

11.3.4.3 Comparison of Cache-Reduction Methods

For one sequence of n tokens, the baseline cache stores $2LnH_{\text{kv}}d_h$ elements for L blocks of H_{kv} key–value heads of width d_h. The table compares the methods of this section on that count. Constants and auxiliary buffers depend on the implementation.

method	reduced factor	stored elements	information given up	training requirement	evidence in this section
GQA/MQA	key–value heads	$2LnH_{\text{kv}}d_h$ with fewer heads	separate keys and values for every query head	train with sharing, or uptrain a checkpoint	character model trained at each H_{kv}; cited large-model ablations
MLA	width per token	about Lnr, plus a small rotary key	keys and values outside an r-dimensional latent	learned compression and reconstruction maps	fixed rank-r basis on GPT-2; cited trained MLA models

method	reduced factor	stored elements	information given up	training requirement	evidence in this section
sliding window with sinks	cached length	$2L(w+s)H_{\text{kv}}d_h$ for window w and s sinks	tokens outside the window and the sinks	train or adapt to the restricted pattern	GPT-2 on one passage; cited StreamingLLM experiments
quantization	bytes per element	$2LnH_{\text{kv}}d_h$ at 2 to 4 bits each instead of 16	precision of every stored number	none: applied to a trained model	an exercise on GPT-2; cited KIVI experiments
linear attention	the token-indexed cache	fixed state matrices, independent of n	exact softmax attention and recall of individual tokens	replace and train the attention mechanism	derivation and experiment in Section 10.5

Quantization combines with every other row, because it changes only the number of bits that store each number. At two to four bits, quantizing keys per channel and values per token keeps generation quality close to that of the full cache (Z. Liu et al. 2024). An exercise tests why this grouping works. Other rows combine too. Hybrid stacks (Chapter 12) pair recurrent layers that keep a fixed state, such as linear attention, with a few full-attention layers, so that only those few layers keep a growing cache.

11.3.5 Summary

Naive generation runs a full forward pass for every new token, so its dense-layer cost is quadratic in the length of the text. A KV cache stores the keys and values of past tokens, which causality leaves unchanged, and makes that cost linear. It needs one preallocated buffer per layer and the position offset of each new token, and it reproduces the logits of the full forward pass to rounding error. The cache holds $2LH_{\text{kv}}d_hnb$ elements for L blocks of H_{kv} key–value heads of width d_h, which for long contexts and many sequences exceeds the weights. Prefill is limited by arithmetic and decode by memory bandwidth, because every decoded token reads all weights and its whole cache. Batching shares the weight reads but not the cache reads. GQA and MQA shrink the cache by sharing key–value heads at little cost in quality, and low-rank latents such as MLA shrink the width stored per token. A sliding window bounds the length, provided it keeps the attention sinks that trained models rely on.

11.3.6 Exercises

1. **Cost of generation.** A model with N parameters reads a prompt of P tokens and then generates T tokens. Count $2N$ dense-layer FLOPs per token of a forward pass.
    1. Count the dense-layer FLOPs of a naive schedule that runs a forward pass over the whole history before every new token, and of `generate_cached`, which runs one prefill over the prompt and then single-token steps. Evaluate the ratio of the two counts for the end-to-end run of Section 11.3.1.2, with $P = 3584$ and $T = 256$.

   2. The end-to-end run is faster with the cache by much less than this ratio of FLOPs. Explain the difference with the arithmetic intensities of the two kinds of forward pass, as Section 11.3.2.1 computes them.
2. code **A missing position offset.** `forward_cached` rotates the queries and keys of new tokens by their absolute positions.
   1. Drop the offset from the RoPE rotation, so that every decoded token is rotated as if it sat at position 0, and rerun the check of Section 11.3.1.2 for `pos='rope'`. Compare the deviation with that of the correct cache.
   2. Take a decoded query at true position t. Determine the relative offset that the faulty cache uses against a prompt key at position j and against the key of an earlier decoded token. Which information about the decoded tokens does the model lose entirely?
   3. Make the corresponding mistake for learned positions, so that every decoded token receives the embedding of position 0. Prefill GPT-2 with the first 512 tokens of the passage `x`, feed the remaining tokens one at a time, and compare the loss on positions 512 onward, the loss that `tail_loss` measures, for the faulty and the correct cache.
3. **Cache of a 70B model.** Llama-2-70B (Touvron et al. 2023b) has 80 layers and 64 query heads of dimension 128, and it uses GQA with 8 key–value heads. Its weights occupy about 140 GB in 16-bit precision, and its cache is stored in 16-bit precision as well.
   1. Use Equation 11.10 to compute the cache bytes per token, and the context length at which the cache of a single sequence reaches the size of the weights.
   2. Repeat both computations for the same model with 64 key–value heads, that is, without sharing, and with a single key–value head (MQA).
   3. The weights are split across two GPUs with 80 GB each. For each of the three configurations, find how many sequences of 4096 tokens fit into the memory that the weights leave free, ignoring activations.
4. code **A basis fitted on the passage itself.** The experiment of Section 11.3.4.1 fits its basis on a second passage.
   1. Fit the basis instead on the keys and values of `x` itself, the passage that `tail_loss` evaluates, and compare the losses at ranks 512, 256, and 128 with those of the basis from `calib` at the same rank.
   2. Explain why this comparison is unfair: which later tokens influence how the key and value of token t are compressed, and why can such an influence lower the loss of predicting token $t + 1$?
   3. A server could instead fit the basis on the prompt, here the first 512 tokens of `x`, and apply it to every later token. Implement this variant and compare it with the other two. Which of the three could a deployed cache use?
5. code **Quantized keys and values.** Store the keys and values of GPT-2 in 4 bits: round each number to one of 16 levels spaced evenly between the minimum and the maximum of its group, where a group is either one channel (one coordinate over all tokens of a head) or one token (all coordinates of one token in a head).
   1. Use `forward_blocks` to evaluate `tail_loss` on `x` for the four combinations of grouping keys and values per channel or per token. Which combination works best?
   2. Measure, in a few blocks, the largest absolute value of each key channel and of each value channel. Use the result to explain your answer to the previous question, and compare it with the choice of Z. Liu et al. (2024) .
   3. How many bits per number does each grouping cost, counting the minimum and the maximum stored with every group in 16 bits, for a context of 1024 tokens?
6. code **Sinks as no-ops.** Xiao et al. (2024) interpret a sink as the position where a head

places attention it does not need. gpt-oss makes this role explicit by giving every head a learned logit a_{sink} in the denominator of its softmax (OpenAI 2025). For the query at position t with scores $a_{t,0}, \dots, a_{t,t}$ on the tokens at positions $0, \dots, t$, the weight on token j is

$$\alpha_{t,j} = \frac{\exp(a_{t,j})}{\exp(a_{\text{sink}}) + \sum_{i=0}^{t} \exp(a_{t,i})}.$$

1. Show that such a head computes the output of ordinary softmax attention over the $t+1$ tokens plus one extra key, whose score is a_{sink} for every query and whose value is the zero vector. Which values can the total weight on the $t+1$ tokens take?
2. GPT-2 has no sink logit, so its weights are those of the formula with $a_{\text{sink}} = -\infty$. Extend `watch` to record, in every block, the norm of the value vector $\mathbf{v}_0$ of token 0 divided by the mean value norm of the other tokens. Record also the norm of $\alpha_{t,0}\,\mathbf{v}_0$ divided by the norm of the head output at position t, averaged over heads and over the queries from position 64 on. In which blocks does token 0 behave like the zero-value key of the previous sub-problem?
3. Repeat the measurement of attention mass on a 1024-token passage that starts 100 tokens later in the book, so that a different token sits at position 0. Does the high mass stay at position 0?

7. **Shared weight reads.** ▯ Use the roofline model of Section 11.3.2.1: a forward pass takes at least its FLOPs divided by the peak rate of 83 TFLOP/s and at least its bytes moved divided by the bandwidth of 1 TB/s. The weights of `model` are stored in fp32.
   1. Ignore the cache. A forward pass of `model` over m new tokens reads every weight once, whether the tokens belong to m sequences of a batch or are m drafted tokens of one sequence. Find the largest m for which the pass takes no longer than a single-token step. Explain why this bound makes it cheap to check several drafted tokens in one pass, as speculative decoding does (Leviathan et al. 2023).
   2. Now include the cache, which each of b sequences at context n reads separately. Derive the arithmetic intensity of one decode step of `model`, counting the attention FLOPs and the cache bytes. Evaluate it at $n = 2048$ for $b = 1$ and in the limit $b \to \infty$, and compare both values with the ridge of about 80 FLOPs per byte. How does the limit change when the H query heads share H_{kv} key–value heads?
8. code **Uptraining to GQA.** ▯ Ainslie et al. (2023) convert a multi-head checkpoint into a grouped one by mean-pooling the key and value projections of the heads in each group, and then continue training for 5 percent of the original number of steps. Apply this recipe to the byte-level model that Section 11.3.3.2 trains.
   1. Train the $H_{\text{kv}} = 8$ configuration for 1,000 steps. Convert it to $H_{\text{kv}} = 2$ by averaging the key and value columns of $\mathbf{W}_{qkv}$ over each group of four consecutive heads, the grouping that `num_kv_heads` uses, and measure the validation loss without further training. Repeat the conversion with the first head of each group in place of the average.
   2. Continue training the mean-pooled model for 50 steps at a small constant learning rate. Compare its validation loss with that of the $H_{\text{kv}} = 8$ model and with that of an $H_{\text{kv}} = 2$ model trained from scratch for 1,050 steps.
9. **Absorbed projections.** ▯ In MLA, every token j of a layer caches one latent vector $\mathbf{c}_j \in \mathbb{R}^r$, and head h reconstructs its key and value as $\mathbf{k}_j^{(h)} = \mathbf{U}_K^{(h)}\mathbf{c}_j$ and $\mathbf{v}_j^{(h)} = \mathbf{U}_V^{(h)}\mathbf{c}_j$, with up-projections $\mathbf{U}_K^{(h)}, \mathbf{U}_V^{(h)} \in \mathbb{R}^{d_h \times r}$ (DeepSeek-AI 2024). Head h has the query $\mathbf{q}_t^{(h)}$ at position t and the attention output $\mathbf{o}_t^{(h)}$, and the layer returns $\sum_h \mathbf{W}_O^{(h)}\mathbf{o}_t^{(h)}$ with $\mathbf{W}_O^{(h)} \in \mathbb{R}^{d \times d_h}$. Ignore the $1/\sqrt{d_h}$ scaling of the scores.

1. Show that a decode step can compute every score and the layer output from the cached latents alone, without forming any key or value, by folding $\mathbf{U}_K^{(h)}$ into the query and $\mathbf{U}_V^{(h)}$ into the output projection.
2. With RoPE, in the notation of Section 10.4, the score becomes $(\mathbf{R}_t\mathbf{q}_t^{(h)})^\top \mathbf{R}_j \mathbf{U}_K^{(h)} \mathbf{c}_j$. Show that folding no longer produces a matrix that is independent of j. MLA therefore appends to every key a rotary part $\mathbf{R}_j\mathbf{k}_j^R$, where $\mathbf{k}_j^R \in \mathbb{R}^{d_R}$ is shared by all heads and cached next to $\mathbf{c}_j$, and it appends a rotary part to every query. Show that the score then splits into a term that admits the folding and a term computed from the cached $\mathbf{k}_j^R$.
3. DeepSeek-V2 uses $H = 128$ heads with $d_h = 128$, $r = 512$, and $d_R = 64$. For one layer and one cached token, compare the number of cached elements and the FLOPs of the scores and weighted sums of a decode step under multi-head attention with cached keys and values and under MLA with folded projections. How many key–value heads would GQA need for a cache of the same size? What does the ratio of FLOPs to cached elements imply for the decode step of Section 11.3.2.1?

10. [extended] **A streaming cache.** ▨▨ The mask-based experiment of Section 11.3.4.2 restricts attention without evicting any entry, and it keeps the original position of every token.
    1. Change `forward_cached` so that the cache of each block is a rolling buffer. It holds the first s entries and the most recent w entries, the current token included, and overwrites the oldest of the remaining entries. Prefill GPT-2 with the first 256 tokens of the passage `x`, feed the remaining tokens one at a time with $w = 256$ and $s = 4$, and confirm that the loss on positions 512 onward matches the mask-based result.
    2. GPT-2 cannot decode beyond position 1023, where its position table ends. A RoPE model can stream indefinitely if the positions of the cached entries are reassigned. Decide what the buffer must store for each entry and which positions the rotation should use, so that every query–key offset stays below $s+w$. Explain why learned position embeddings do not admit the same reassignment.
    3. Train the byte-level `d2l.GPT` of Section 11.3.3.2, with its default RoPE attention, at context 128. Stream the first 10,000 bytes of the validation text, `data.val` read in order, through your buffer with $s + w = 128$. Compare the mean loss for $s = 4$ and for $s = 0$ with that of a full forward pass over the most recent 128 bytes at every step.

11.4 Encoders, Decoders, and Cross-Attention

The GPT of Section 11.2 reads text from left to right: its causal mask lets each position see only what came before it. This is what next-token prediction needs, but many tasks are not next-token prediction. To fill the gap in "the ___ barked at the mail carrier", a reader uses the words after the gap as much as those before it. To translate a sentence, a translator reads all of it before writing the first word of the translation. What does a transformer gain when it may see its whole input, and how does it produce one sequence from another?

Both questions concern two choices that the GPT fixed: the mask, and the sources of the queries, keys, and values. Removing the mask gives an *encoder*, which reads in both directions and produces contextual representations. Joining a masked stack to an unmasked one through the cross-attention of Section 10.3 gives the *encoder–decoder*, the form in which

transformers began, as a model for machine translation (Vaswani et al. 2017). We build both from the same `d2l.TransformerBlock`, measure how much the text after a token lowers the loss of predicting it, and check whether a learned cross-attention map reproduces a known alignment. We then turn to learned queries that do not come from an input sequence. Such queries connect a computation of fixed size to data of variable size, as in Perceiver-style and vision–language models.

11.4.1 Encoder, Decoder, and Encoder–Decoder Architectures

A transformer block maps a sequence of d-dimensional vectors to a sequence of the same shape. It leaves two questions open: which positions may attend to which, and where the keys and values come from. Three common wirings answer these questions. Figure 11.6 draws them in the convention we will use for every attention map in this section: queries down the rows, keys along the columns.

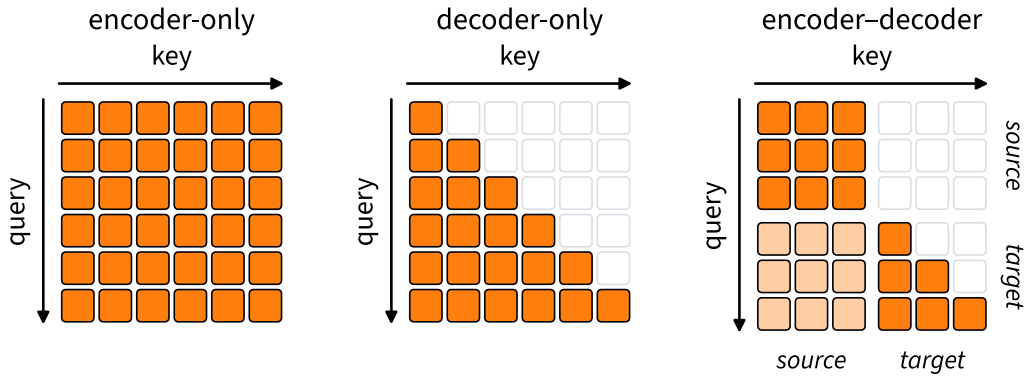


Figure 11.6: The three wirings and their attention patterns (query position runs downward, key position rightward; a filled cell means the query may attend to that key). Encoder-only: every token attends to every token. Decoder-only: each token attends only to itself and its past. Encoder–decoder: the source attends to itself bidirectionally, while each target token attends to the full source through cross-attention (light cells) and to its own past causally.

Encoder-only. Drop the mask. Every token attends to every other, so each output vector summarizes the *whole* input as seen from its position. Such a model does not directly implement the left-to-right autoregressive factorization: with the future visible, next-token prediction reduces to copying. (Masked models can still generate text by iterative re-masking, the idea that text diffusion models develop.) But the encoder is the strongest way to *represent* an input, and one representation per token is what classification, retrieval, and tagging consume. BERT is this wiring pretrained on text (Devlin et al. 2018); the vision transformer of Section 11.5 is the same wiring over image patches.

Decoder-only. Keep the causal mask everywhere: this is the GPT of Section 11.2. Each token predicts its successor, and generation is the training objective run forward. One stack is trained with one objective, next-token prediction on raw text, and the same stack serves generation. Large enough models of this kind perform tasks that are specified only in the prompt (Brown et al. 2020). We do not rebuild this wiring here. In Figure 11.6 it is the pattern against which the other two are compared.

Encoder–decoder. The original transformer (Vaswani et al. 2017) combines the two: a

bidirectional encoder reads the source sequence, and a causal decoder generates the target while *cross-attending* into the encoder's output. This is the cross-attention of Section 10.3, with queries from the target stream and keys and values from the source. The pattern in Figure 11.6 shows the division: a full square for the source, a full rectangle for target-to-source cross-attention, a triangle for the target's own past. The wiring fits tasks whose input is fully known before generation starts. Machine translation was the founding application. T5 pretrained the architecture on span reconstruction and recast a broad family of tasks as text-to-text (Raffel et al. 2020), and BART pretrained it on denoising corrupted text (Lewis et al. 2019). In speech recognition, Whisper's encoder reads an entire audio clip bidirectionally while a text decoder cross-attends into it (Radford et al. 2023).

The rest of this section builds the two wirings that Section 11.2 did not: first the encoder, then the encoder–decoder.

11.4.2 Bidirectional Context for Masked-Token Prediction

11.4.2.1 A Bidirectional Encoder

The encoder differs from the `CharLM` of Section 11.1 in one respect: its blocks receive no mask, so every token attends to every token. Attention is permutation equivariant (Section 10.4), so the model must still supply positions, and the learned position table stays.

```
class TransformerEncoder(nn.Module):
    """Bidirectional encoder: embeddings plus unmasked pre-norm blocks."""
    def __init__(self, vocab_size, num_hiddens=128, num_heads=4, num_blks=4,
                 max_len=64):
        super().__init__()
        self.token_emb = nn.Embedding(vocab_size, num_hiddens)
        self.pos_emb = nn.Embedding(max_len, num_hiddens)
        for emb in (self.token_emb, self.pos_emb):
            nn.init.normal_(emb.weight, std=0.02)
        self.blks = nn.ModuleList([
            d2l.TransformerBlock(num_hiddens, num_heads)
            for _ in range(num_blks)])
        self.norm = nn.RMSNorm(num_hiddens, eps=1e-5)

    def forward(self, X):
        H = self.token_emb(X) + self.pos_emb(
            torch.arange(X.shape[1], device=X.device))
        for blk in self.blks:
            H = blk(H)  # no mask: every token attends to every token
        return self.norm(H)
```

11.4.2.2 The Masked-Token Objective

This model cannot be trained on next-token prediction. With the future visible, position t can read token $t+1$ directly, and the loss collapses without teaching anything. The objective must hide what it asks for. *Masked language modeling* (Devlin et al. 2018) replaces a random subset of tokens with a special `<mask>` symbol and asks the model to reconstruct them from everything that remains:

$$\max \sum_{t \in \mathcal{M}} \log p\left(x_t \mid \mathbf{x}_{\setminus \mathcal{M}}\right), \tag{11.12}$$

where $\mathcal{M}$ is the set of masked positions. A prediction at a masked position draws on context from *both* sides, which is what removing the causal mask provides. We train this objective in miniature on `d2l.WikiText103`, the bytes of the Wikipedia articles of Section 9.5.2. In each window 15% of the bytes are masked, and `<mask>` gets one extra embedding row after the 256 byte values. The labels are the masked bytes, and -100 at every other position; `d2l.train_steps` of Section 9.5 leaves positions with this label out of the cross-entropy, so it trains on Equation 11.12, averaged over the masked positions. `MaskedLM` adds an output head, tied to the embedding as in Section 11.1.

```
data = d2l.WikiText103(batch_size=64, num_steps=64)
MASK = len(data.vocab)  # id of the extra <mask> embedding row

def mask_tokens(X, p=0.15):
    mask = torch.rand(X.shape, device=X.device) < p
    return torch.where(mask, MASK, X), mask

class MaskedBytes:
    """The windows of data with masked bytes; labels -100 add no loss."""
    def __init__(self, data):
        self.data = data

    def train_dataloader(self):
        for X, _ in self.data.train_dataloader():
            Xm, mask = mask_tokens(X)
            yield Xm, torch.where(mask, X, -100)

class MaskedLM(nn.Module):
    """An encoder with an output head tied to its token embedding."""
    def __init__(self, encoder):
        super().__init__()
        self.encoder = encoder

    def forward(self, X):
        return F.linear(self.encoder(X), self.encoder.token_emb.weight)

device = d2l.try_gpu()
model = MaskedLM(TransformerEncoder(len(data.vocab) + 1)).to(device)
optimizer = torch.optim.AdamW(model.parameters(), lr=1e-3,
                              weight_decay=0.0)
losses = d2l.train_steps(model, MaskedBytes(data), optimizer, 2000)
print('masked loss at step 500/1000/2000: ' + '/'.join(
    f'{sum(losses[k-50:k]) / 50:.2f}' for k in (500, 1000, 2000)))
```

```
masked loss at step 500/1000/2000: 3.12/2.08/1.63
```

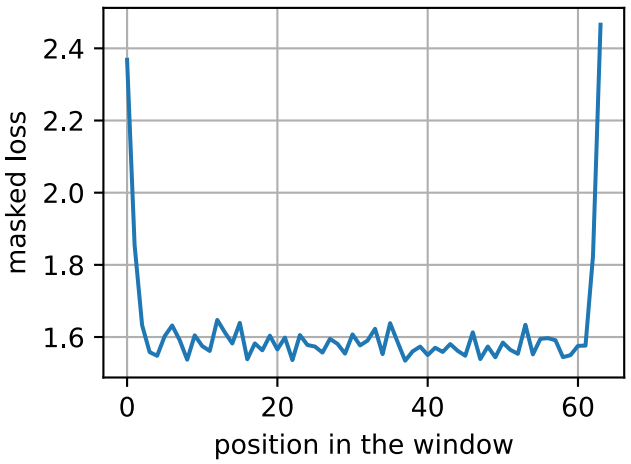
Two thousand steps bring the masked loss far below 3.17 nats, the unigram entropy of the byte frequencies, and the model recovers about half of the masked bytes exactly. Binning this average by position shows how much each amount of available context contributes.

11.4.2.3 Loss by Available Context

Our windows are 64 bytes long, so every batch contains a comparison. A masked byte in the interior has context on both sides, while one at position 0 or 63 sees only one side. The last position has the context that a causal language model has at every step. The model and the objective are the same at every position, so binning the validation loss by position measures what context from the second side contributes.

```
model.eval()
num_steps = data.num_steps  # Window length: 64 positions
pos_loss = torch.zeros(num_steps, device=device)
pos_cnt = torch.zeros(num_steps, device=device)
correct = torch.zeros((), device=device)
with torch.no_grad():
    for X, _ in data.val_dataloader():
        X = X.to(device)
        Xm, mask = mask_tokens(X)
        logits = model(Xm)
        loss = F.cross_entropy(logits.transpose(1, 2), X, reduction='none')
        pos_loss += (loss * mask).sum(0)
        pos_cnt += mask.sum(0)
        correct += ((logits.argmax(-1) == X) & mask).sum()
print(f'masked accuracy {correct / pos_cnt.sum():.2f}')
pos_loss = (pos_loss / pos_cnt).cpu()
print(f'loss at position 0: {pos_loss[0]:.2f}, at position '
      f'{num_steps - 1}: {pos_loss[-1]:.2f}, interior mean: '
      f'{pos_loss[num_steps // 4:-num_steps // 4].mean():.2f}')
d2l.plot(torch.arange(num_steps), pos_loss, 'position in the window',
         'masked loss')
```

```
masked accuracy 0.52
loss at position 0: 2.37, at position 63: 2.47, interior mean: 1.57
```



The profile is flat across the interior and rises by about half or more at the two one-sided edges. The difference is the contribution of the second side: conditioning on both sides of a byte lowers its loss by about a third relative to one side. The two edges cost about the same,

because in text what follows a gap constrains it about as strongly as what precedes it. We can also read the predictions directly. We mask one byte of a sentence from the validation text and list the model's three most probable fills:

```
snippet = 'Homarus gammarus , known as the European lobster or common '
ids = torch.tensor(data.vocab.encode(snippet), device=device)[None]
for pos in (11, 22, 28):
    Xm = ids.clone()
    Xm[0, pos] = MASK
    with torch.no_grad():
        logits = model(Xm)
    probs = F.softmax(logits[0, pos], -1)
    top = probs.topk(3)
    shown = snippet[:pos] + '_' + snippet[pos + 1:]
    print(f'{shown[:40]!r}... -> '
          + ', '.join(f'{data.vocab.decode([int(i)])!r} {p:.2f}'
                      for p, i in zip(top.values, top.indices)))
```

```
'Homarus gam_arus , known as the European'... -> 'p' 0.34, ' ' 0.32, 'e' 0.10
'Homarus gammarus , kno_n as the European'... -> 'w' 0.80, 'i' 0.04, 'o' 0.04
'Homarus gammarus , known as _he European'... -> 't' 0.95, 'T' 0.02, 'w' 0.01
```

In "known as _he European", the model puts most of its probability on "t". The left context "known as" allows many words, while the right-hand context ("he European") leaves essentially one, and a left-to-right model never sees that context. Scaled up to subword tokens, sentence pairs, gigabytes of text, and a fine-tuning recipe per downstream task, this experiment is BERT (Devlin et al. 2018). The Language Models part covers its pretraining and fine-tuning in full (Chapter 43). Encoders have since adopted the block internals of this chapter: ModernBERT keeps the encoder-only wiring and uses RoPE, a gated FFN, alternating local and global attention, and a context of 8,192 tokens (Warner et al. 2024).

11.4.3 Cross-Attention in an Encoder–Decoder

11.4.3.1 A Task with Known Alignment

An encoder–decoder is useful when the input and output are different sequences. Its signature component, cross-attention, is usually illustrated on machine translation, but a trained translation model gives us no ground truth to check its attention against. So we choose a task whose correct alignment is known by construction: *reverse a random string*. Target position t must copy source position $n-1-t$. The letters are drawn independently at random, and the only path from the source to the decoder runs through cross-attention. If the mechanism works and the model solves the task directly, its attention map should be the anti-diagonal, a prediction we can check. Random strings also mean unlimited fresh data: like the GPT of Section 11.2, which almost never reads the same passage twice, this model never sees the same example twice.

```
V, T = 16, 12   # 16 letters, strings of length 12
BOS = V         # decoder start token

def sample_batch(batch_size, device):
    src = torch.randint(0, V, (batch_size, T), device=device)
```

```
    tgt = src.flip(1)
    dec_in = torch.cat([torch.full((batch_size, 1), BOS, device=device),
                        tgt[:, :-1]], 1)
    return src, dec_in, tgt

def to_str(ids):
    return ''.join(chr(ord('a') + int(i)) for i in ids)

src, dec_in, tgt = sample_batch(1, device)
print(f'source {to_str(src[0])!r} -> target {to_str(tgt[0])!r}')
```

```
source 'edlfpbeclopl' -> target 'lpolcebpflde'
```

As in Section 11.2, training uses teacher forcing: the decoder input is the target shifted right behind a `<bos>` token, so every position learns to predict its successor in parallel.

11.4.3.2 The Decoder Block

A decoder block is the transformer block plus one sublayer. Between the causal self-attention and the FFN sits cross-attention: queries from the target's residual stream, keys and values from the encoder output. It follows the same pre-norm arrangement as every block in this chapter: each sublayer reads the stream through a normalization and adds its result back. The two attention sites are masked differently. Self-attention stays causal, because the target is being generated, while cross-attention is unmasked, because the source is fully known before generation starts. Both masks are optional arguments, so a padded batch can pass its own (Section 11.4.3.3). By default, self-attention gets `d2l.causal_mask` and cross-attention no mask.

```
class DecoderBlock(nn.Module):
    """Pre-norm decoder block: causal self-attention, cross-attention,
    FFN."""
    def __init__(self, num_hiddens, num_heads):
        super().__init__()
        self.norm1 = nn.RMSNorm(num_hiddens, eps=1e-5)
        self.norm2 = nn.RMSNorm(num_hiddens, eps=1e-5)
        self.norm3 = nn.RMSNorm(num_hiddens, eps=1e-5)
        self.self_attention = d2l.MultiHeadAttention(num_hiddens, num_heads)
        self.cross_attention = d2l.MultiHeadAttention(num_hiddens, num_heads)
        self.ffn = d2l.FeedForward(num_hiddens)

    def forward(self, X, enc, mask=None, cross_mask=None):
        if mask is None:  # causal self-attention
            mask = d2l.causal_mask(X.shape[1], device=X.device)
        Y = self.norm1(X)
        X = X + self.self_attention(Y, Y, Y, mask)
        X = X + self.cross_attention(self.norm2(X), enc, enc, cross_mask)
        return X + self.ffn(self.norm3(X))
```

The full model wires a `TransformerEncoder` (the class from the previous section, reused unchanged) to a stack of decoder blocks with its own embeddings and output head. `EncoderDecoder` returns the logits; each attention module keeps the weights of its last call in `attention_weights`. The method `decode` runs the decoder alone, so that generation can run the

encoder once and then only the decoder for each new token.

```
class EncoderDecoder(nn.Module):
    """A causal decoder cross-attending into a bidirectional encoder."""
    def __init__(self, vocab_size, num_hiddens=128, num_heads=4,
                 num_blks=1, max_len=64):
        super().__init__()
        self.encoder = TransformerEncoder(vocab_size, num_hiddens,
                                          num_heads, num_blks, max_len)
        self.tgt_emb = nn.Embedding(vocab_size, num_hiddens)
        self.pos_emb = nn.Embedding(max_len, num_hiddens)
        for emb in (self.tgt_emb, self.pos_emb):
            nn.init.normal_(emb.weight, std=0.02)
        self.blks = nn.ModuleList([DecoderBlock(num_hiddens, num_heads)
                                   for _ in range(num_blks)])
        self.norm = nn.RMSNorm(num_hiddens, eps=1e-5)
        self.head = nn.Linear(num_hiddens, vocab_size, bias=False)

    def decode(self, dec_in, enc):
        H = self.tgt_emb(dec_in) + self.pos_emb(
            torch.arange(dec_in.shape[1], device=dec_in.device))
        for blk in self.blks:
            H = blk(H, enc)
        return self.head(self.norm(H))

    def forward(self, src, dec_in):
        return self.decode(dec_in, self.encoder(src))
```

11.4.3.3 Assembling the Masks

Our reversal task avoids padding by construction: every string has the same length, and the only mask in use is the decoder's causal one. A real batch holds sequences of different lengths, padded on the right to a rectangle, and it needs one mask per attention site. Each is composed from the two primitives of Section 10.2, the padding mask `d2l.padding_mask` built from valid lengths and the causal triangle `d2l.causal_mask`, combined by logical AND under broadcasting. Encoder self-attention masks source padding. Decoder self-attention during teacher-forced training needs the causal triangle *and* target padding on the key side (padded query rows compute outputs that the loss ignores). Cross-attention masks source padding again: the target may be mid-generation, but the source it reads is fully known. The cell below assembles all three for a toy ragged batch as boolean arrays of shape (batch, queries, keys). `d2l.MultiHeadAttention` takes such masks directly, as do the fused kernels of Section 10.5.

```
src_len, tgt_len = torch.tensor([3, 5]), torch.tensor([3, 4])
S, T_dec = int(src_len.max()), int(tgt_len.max())
B = len(src_len)
src_valid = d2l.padding_mask(src_len, S)         # (B, 1, S)
tgt_valid = d2l.padding_mask(tgt_len, T_dec)     # (B, 1, T)
enc_self = src_valid.expand(-1, S, -1)           # (B, S, S): source padding
dec_self = d2l.causal_mask(T_dec) & tgt_valid    # (B, T, T): causal, padding
cross = src_valid.expand(-1, T_dec, -1)          # (B, T, S): source padding
```

```
for name, m in (('encoder self', enc_self), ('decoder self', dec_self),
                ('cross', cross)):
    print(f'{name}-attention mask, sequence 0 (1 = may attend):')
    print(m[0].int().numpy())
```

```
encoder self-attention mask, sequence 0 (1 = may attend):
[[1 1 1 0 0]
 [1 1 1 0 0]
 [1 1 1 0 0]
 [1 1 1 0 0]
 [1 1 1 0 0]]
decoder self-attention mask, sequence 0 (1 = may attend):
[[1 0 0 0]
 [1 1 0 0]
 [1 1 1 0]
 [1 1 1 0]]
cross-attention mask, sequence 0 (1 = may attend):
[[1 1 1 0 0]
 [1 1 1 0 0]
 [1 1 1 0 0]
 [1 1 1 0 0]]
```

In the three grids of sequence 0 (source length 3 of 5, target length 3 of 4), the encoder square and the cross rectangle both blank the same two padded source columns, and the decoder triangle loses its last column to target padding. These grids are what an encoder–decoder assembles for every padded batch it trains on. To see the model consume them, we pass `dec_self` and `cross` to a fresh `DecoderBlock` on random inputs of width 16 and print the cross-attention weights of sequence 0 in one head, in percent:

```
blk = DecoderBlock(16, num_heads=2)
X, enc = torch.randn(B, T_dec, 16), torch.randn(B, S, 16)
with torch.no_grad():
    blk(X, enc, dec_self, cross)
weights = blk.cross_attention.attention_weights[0, 0]  # (queries, keys)
print((100 * weights).round().int().numpy())
```

```
[[43 20 37  0  0]
 [26 43 31  0  0]
 [35 42 23  0  0]
 [37 20 42  0  0]]
```

Each row spreads its weight over the three real source positions, and the two padded columns receive exactly zero. None of this arises in the reversal model: `sample_batch` produces no padding, so the decoder keeps its default causal mask and the cross-attention runs unmasked.

11.4.3.4 Training and Decoding

One encoder block, one decoder block, and a few hundred steps on freshly sampled batches suffice. Over 600 steps of 128 strings the model reads about a million source characters and never sees a string twice. The optimizer is `d2l.LMOptimizer` of Section 11.2, with its gradient clipping but without warmup or weight decay. Once the loss is near zero, Adam at a constant

learning rate makes it jump now and then, to tenths of a nat or more, for about twenty steps. A jump near the end of the run leaves a model that fails on many strings. The cosine decay of the learning rate to zero prevents this.

```
seq2seq = EncoderDecoder(V + 1).to(device)
optimizer = d2l.LMOptimizer(seq2seq, lr=1e-3, num_steps=600, warmup=0,
                            weight_decay=0.0)
losses = []
for step in range(600):
    src, dec_in, tgt = sample_batch(128, device)
    logits = seq2seq(src, dec_in)
    loss = F.cross_entropy(logits.flatten(0, 1), tgt.flatten())
    optimizer.zero_grad()
    loss.backward()
    optimizer.step()
    losses.append(loss.detach())  # Stays on the device
losses = torch.stack(losses).tolist()  # One transfer at the end
print('loss at step 100/200/400/600: ' + '/'.join(
    f'{sum(losses[k-20:k]) / 20:.3f}' for k in (100, 200, 400, 600)))
```

```
loss at step 100/200/400/600: 0.016/0.003/0.000/0.000
```

Decoding runs the encoder once and the decoder autoregressively: feed <bos>, take the argmax, append, repeat. On a thousand fresh strings the model reverses essentially every one exactly.

```
seq2seq.eval()
with torch.no_grad():
    src, _, tgt = sample_batch(1000, device)
    enc = seq2seq.encoder(src)  # the encoder runs once
    out = torch.full((1000, T + 1), BOS, device=device)
    for t in range(T):
        logits = seq2seq.decode(out[:, :t + 1], enc)
        out[:, t + 1] = logits[:, -1].argmax(-1)
pred = out[:, 1:]
print(f'exact match on 1000 fresh strings: '
      f'{(pred == tgt).all(-1).float().mean():.3f}')
print(f'source {to_str(src[0])!r} -> predicted {to_str(pred[0])!r}')
```

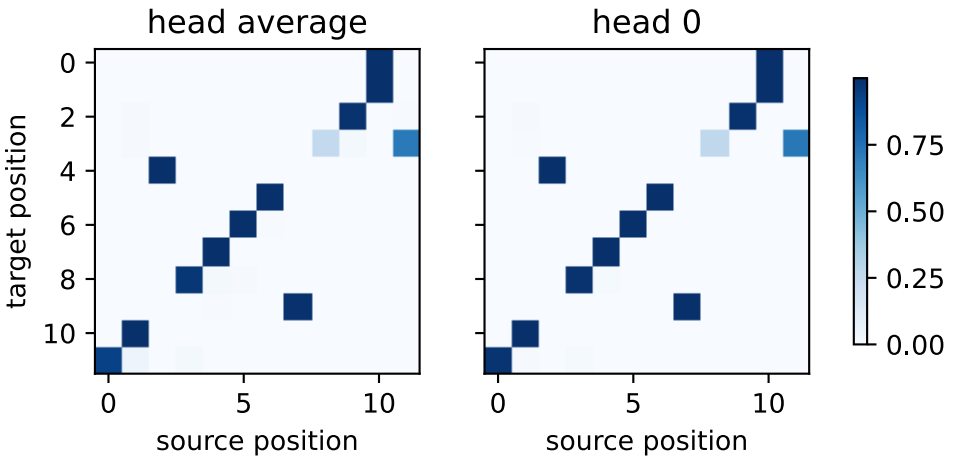
```
exact match on 1000 fresh strings: 1.000
source 'emgjcfnbagno' -> predicted 'ongabnfcjgme'
```

11.4.3.5 Reading the Alignment

A low loss shows that the model reverses strings, but not how. Because the task dictates which source position each target position needs, we can check whether cross-attention finds it. We run a batch through the model, read the cross-attention weights of the decoder block, of shape (batch, heads, target positions, source positions), and compare each target position's attention with that alignment. Besides the head average we report how much weight each head puts on the true source position, and we plot the average next to the head that puts the most weight there.

```
with torch.no_grad():
    src, dec_in, tgt = sample_batch(64, device)
    seq2seq(src, dec_in)
w = seq2seq.blks[0].cross_attention.attention_weights  # (64, 4, T, T)
want = torch.arange(T - 1, -1, -1, device=device)  # row t -> column T-1-t
avg = w.mean(1)                                    # average over heads
cols = avg.argmax(-1)                  # the source column each row reads
hit = (cols == want).float().mean()
same = (cols == cols[0]).float().mean()  # as in the first string
on_target = w[..., torch.arange(T), want].mean((0, 2))  # one per head
print(f'head average: argmax on the true source position in '
      f'{100 * hit:.0f}% of rows, mean weight there {on_target.mean():.2f}')
print('weight on the true position, per head:',
      ' '.join(f'{v:.2f}' for v in on_target))
print(f'argmax in the same column for every string: {100 * same:.0f}% '
      'of rows')
h = int(on_target.argmax())
d2l.show_heatmaps(torch.stack([avg[0], w[0, h]]).cpu()[None],
                  xlabel='source position', ylabel='target position',
                  titles=['head average', f'head {h}'], figsize=(6, 3),
                  cmap='Blues')
```

```
head average: argmax on the true source position in 67% of rows, mean weight there 0.68
weight on the true position, per head: 0.68 0.66 0.68 0.68
argmax in the same column for every string: 100% of rows
```



Whether the maps show the anti-diagonal predicted by the third panel of Figure 11.6 depends on the run. In some runs they do: averaged over heads, the argmax lands on the true source position in almost every row, with most of the softmax mass concentrated there. In others it lands there in only about half of the rows or fewer, even in none, although the model still reverses essentially every string. The last line of the output shows what all runs share. Each target position reads mainly from one source column, the same for most strings and in many runs for nearly all, so the decoder does retrieve by position, as the task requires. But the encoder has a position table of its own, and its self-attention can move every letter to another position before the decoder reads it. Any such rearrangement, undone by the

cross-attention, reverses the string just as well, and training settles on one of them at random. The alignment is then split between two attention layers, and the cross-attention map shows only the second half. As Section 10.3 noted, readable attention patterns are the exception rather than the rule. Even in this one-block model a single map can mislead, and with more depth and more heads the maps spread out further, as exercise 4 shows.

11.4.4 Cross-Attention as Interface

11.4.4.1 Learned Queries

In the decoder, the cross-attention queries came from the target stream. Nothing in the mechanism requires that source. A set of M query vectors can instead be *learned parameters*, a fixed array that exists before any input arrives. Cross-attending it into an input of length N costs $O(MN)$. The quadratic term of Section 10.5 never appears, because the M latents only ever self-attend among themselves, at $O(M^2)$. This is the *Perceiver* (Jaegle et al. 2021): a fixed-size latent bottleneck that reads arbitrarily long, arbitrarily structured inputs through cross-attention, sketched in Figure 11.7.

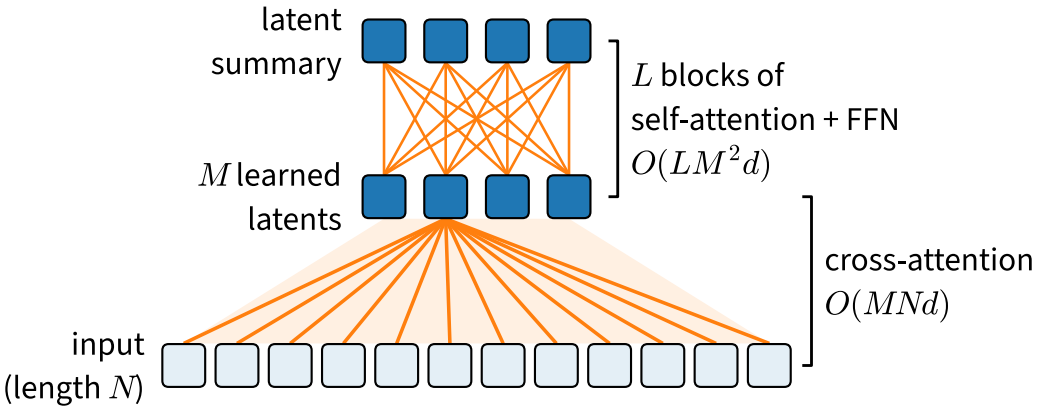


Figure 11.7: A latent bottleneck with input length N, M learned latents, width d, and L latent blocks. Every latent reads every input (shaded band); the read of one latent is drawn in full. One cross-attention read has mixing cost $O(MNd)$; each subsequent latent self-attention layer has mixing cost $O(M^2d)$, for $O(LM^2d)$ across the stack. These terms hold batch size, head count, and width fixed and omit projections and FFNs.

A minimal version needs a parameter array, one cross-attention, and a stack of ordinary transformer blocks over the latents:

```
class PerceiverEncoder(nn.Module):
    """M learned latents cross-attend into the input, then process among
    themselves."""
    def __init__(self, num_latents, num_hiddens, num_heads=4, num_blks=2):
        super().__init__()
        self.latents = nn.Parameter(
            0.02 * torch.randn(num_latents, num_hiddens))
        self.norm_q = nn.RMSNorm(num_hiddens, eps=1e-5)
```

```
        self.cross_attention = d2l.MultiHeadAttention(num_hiddens, num_heads)
        self.blks = nn.ModuleList([
            d2l.TransformerBlock(num_hiddens, num_heads)
            for _ in range(num_blks)])
        self.norm = nn.RMSNorm(num_hiddens, eps=1e-5)

    def forward(self, X):
        Z = self.latents.expand(X.shape[0], -1, -1)
        Z = Z + self.cross_attention(self.norm_q(Z), X, X)  # O(MN)
        for blk in self.blks:
            Z = blk(Z)                                      # O(M^2)
        return self.norm(Z)
```

We read a sequence of 4,096 random feature vectors into $M = 64$ latents:

```
perceiver = PerceiverEncoder(num_latents=64, num_hiddens=256).to(device)
perceiver.eval()
X = torch.randn(1, 4096, 256, device=device)
print('input', tuple(X.shape), '-> latent summary',
      tuple(perceiver(X).shape))
```

```
input (1, 4096, 256) -> latent summary (1, 64, 256)
```

Whatever the input length, the output is $M = 64$ vectors. The input here is a raw feature sequence rather than token embeddings, because the encoder does not depend on what the input is: text, audio frames, image pixels, or a concatenation of all three.

Learned queries can also set the shape of the output. Perceiver IO (Jaegle et al. 2022) adds an array of output queries that cross-attends *out of* the latent summary, so the output has one vector per output query, whatever the input: one query for a classification label, one per pixel for optical flow, one per audio sample for a waveform. Input length, number of latents, and output shape become three independent choices, and the latent transformer keeps its fixed cost.

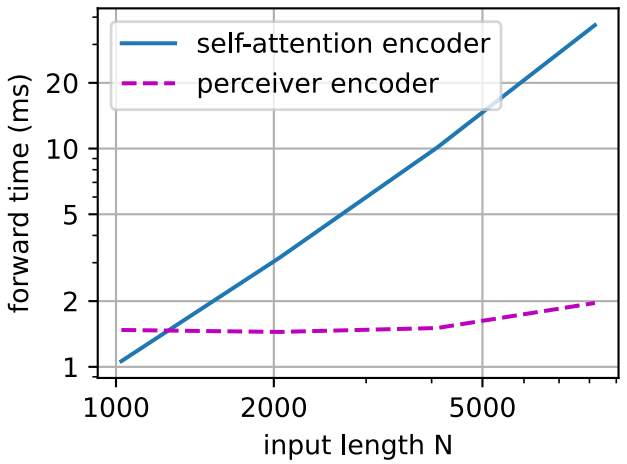
11.4.4.2 The Cost Curve

We check the shape of this cost. We time the Perceiver against the direct alternative (the same two transformer blocks applied to the full input sequence) as N doubles at fixed $M = 64$. The timer is `d2l.Benchmark` of Section 13.1: it runs the function a few times to warm up, waits for the device to finish, and reports the average time of further calls.

```
# The comparison: the same two blocks over the full input, O(N^2)
full = nn.Sequential(*[d2l.TransformerBlock(256, num_heads=4)
                       for _ in range(2)]).to(device).eval()
lengths, t_full, t_perc = [1024, 2048, 4096, 8192], [], []
with torch.no_grad():
    for N in lengths:
        X = torch.randn(1, N, 256, device=device)
        t_full.append(1e3 * d2l.Benchmark(lambda: full(X), repeats=20).time)
        t_perc.append(1e3 * d2l.Benchmark(lambda: perceiver(X),
                                          repeats=20).time)
```

```
        print(f'N={N:5d}: self-attention {t_full[-1]:6.2f} ms, '
              f'perceiver {t_perc[-1]:5.2f} ms')
    d2l.plot(lengths, [t_full, t_perc], 'input length N', 'forward time (ms)',
             xscale='log', yscale='log',
             legend=['self-attention encoder', 'perceiver encoder'])
```

```
N= 1024: self-attention   1.06 ms, perceiver  1.48 ms
N= 2048: self-attention   3.15 ms, perceiver  1.45 ms
N= 4096: self-attention  10.10 ms, perceiver  1.51 ms
N= 8192: self-attention  36.76 ms, perceiver  1.96 ms
```



Each doubling of N eventually multiplies the self-attention encoder's time by about four, the signature of an N^2 term taking over. The Perceiver's time grows far more slowly over this range: its $O(MN)$ cross-attention grows linearly, but the fixed $O(M^2)$ latent processing dominates. By $N = 8192$ the gap exceeds an order of magnitude. At $N = 1024$ the two encoders take comparable time, because the Perceiver's fixed latent cost is then a large fraction of its total. A latent bottleneck is the better choice when the input is long and a fixed-size summary suffices.

The three wirings differ in what they produce. An encoder produces one representation per input token, which classification and retrieval consume. An encoder–decoder generates a sequence conditioned on a separate input that is fully known in advance. A decoder-only model continues a sequence, with one stack for training and for generation. Learned queries are a fourth use of cross-attention: they decouple the cost of attention from the length of the input.

11.4.5 Summary

A transformer block supports encoder-only, decoder-only, and encoder–decoder architectures. Removing the causal mask produces an encoder that represents each token using context from both directions. In our byte-level experiment, positions with context on both sides have about a third less loss than positions with only left context. An encoder–decoder connects an unmasked encoder to a causal decoder through cross-attention; on a reversal task, it reads each target position mainly from one fixed source position, the anti-diagonal

of the task or, when the encoder has first moved the letters, a rearrangement of it. Cross-attention can also use a learned latent array as its queries. Reading a length-N input into M latents costs $O(MN)$ rather than $O(N^2)$, and for fixed M its measured runtime grows far more slowly with N than that of full self-attention. Encoder-only models are common for retrieval and classification, encoder–decoders for translation and speech, and decoder-only models for general autoregressive generation.

11.4.6 Exercises

1. **Prefix language model.** A *prefix language model* wires a single stack in a fourth way. The first m positions, which hold the source, attend to one another bidirectionally, and each of the n target positions that follow attends to the whole source and causally to the target positions up to itself. Raffel et al. (2020) compare this wiring with the encoder–decoder.
    1. Draw its attention pattern for $m = 3$ and $n = 3$ in the convention of Figure 11.6. Number all $m + n$ positions from 0 and write the mask entry for query t and key j (true when t may attend to j) as a formula in t, j, and m. Build the mask from `d2l.causal_mask` and one block that unmasks the source, and check it against your drawing.
    2. In the encoder–decoder, the cross-attention of every decoder block reads the output of the last encoder block. Which representations of the source does a target token read in block l of a prefix language model? On which tokens can a source token's representation depend in each design?
    3. Let every block have c parameters, each applied once to every token that the block processes, and ignore the cross-attention sublayer and the attention scores. Compare the parameter count and the training FLOPs per example of an encoder–decoder with L encoder blocks and L decoder blocks against prefix language models with L blocks and with $2L$ blocks. Which prefix model matches the encoder–decoder in parameters, and which in compute?
2. code **The masking recipe.** `mask_tokens` masks 15% of the bytes and writes `<mask>` at every masked position. Evaluate every variant below with the evaluation of Section 11.4.2.3, so that only the training recipe changes.
    1. BERT replaces a chosen token with `<mask>` 80% of the time, with a random token 10% of the time, and leaves it unchanged 10% of the time (Devlin et al. 2018), in part because `<mask>` never appears when the pretrained encoder reads real text. Implement this rule, retrain, and compare the masked accuracy with that of the original recipe. Then feed both encoders unmodified validation windows and measure the loss of predicting each input byte at its own position. Explain what each of the two 10% cases teaches the model.
    2. Wettig et al. (2023) find that masking rates well above 15% can work better for large models. Retrain with the rates $p \in \{0.05, 0.15, 0.4\}$ for the same number of steps and compare the masked loss at evaluation. Explain the result by what a higher rate adds to each training window and what it removes.
3. **Padding masks of a ragged batch.** The cell of Section 11.4.3.3 builds the three masks for a toy batch with source lengths $(3, 5)$ and target lengths $(3, 4)$, padded on the right.
    1. Compare `dec_self` with the causal triangle alone and name the rows that differ. Would replacing `dec_self` by the causal triangle change the training loss of a decoder with any number of blocks? Explain. Then decide whether the encoder output at a padded source position can reach the loss.
    2. Now pad the batch on the left, so that every sequence ends at the last position.

Which of the three masks change, and does the causal triangle still suffice for decoder self-attention? `d2l.padding_mask` marks the first `valid_lens` keys of each sequence as valid. Which of the three masks can it still build?

3. What goes wrong with the learned position tables of `EncoderDecoder` under left padding unless the position indices change as well?

4. code **Depth, heads, and readable maps.** The alignment check of Section 11.4.3.5 reads one decoder block with four heads.
   1. Retrain `EncoderDecoder` once with `num_blks=2` and once with `num_heads=8`, in each case until the exact match on 1000 fresh strings exceeds 0.99. Rerun the alignment check for every decoder block, and plot the map of every head.
   2. Give `EncoderDecoder` separate depths for its encoder and its decoder. Repeat the check with two encoder blocks and one decoder block, and with one encoder block and two decoder blocks. Compare the maps of both models with those of the one-block model.
   3. Follow each letter back through the encoder: multiply the head-averaged cross-attention weights by the head-averaged weights of the encoder's self-attention, and compute the hit rate of the product. Over several runs, does the product recover the anti-diagonal where the cross-attention alone does not? When does the residual stream make this tracing fail? Reconcile your findings with the cautions of Section 10.3 about reading attention maps.

5. code **Several masks at once.** ⊠ The objective Equation 11.12 scores each masked position separately, so the encoder returns one distribution per masked position, conditioned only on the bytes that remain visible. Filling all masked positions from one forward pass therefore chooses each byte independently of the others. Iterative schemes such as Mask-Predict (Ghazvininejad et al. 2019) keep only their most confident predictions and predict the remaining positions again.
   1. Construct a distribution over the two-byte fills of a gap for which the pair of per-position argmaxes has probability zero. Show that filling the more confident position first and then filling the other from its exact conditional distribution always yields a pair of positive probability.
   2. In each validation window, mask one run of k adjacent bytes, placed at random with at least 8 unmasked bytes on either side, and nothing else. For $k \in \{1, 2, 4, 8\}$, measure the byte accuracy of the parallel fill and of an iterative fill that commits one byte per forward pass, always at the masked position whose top probability is largest.
   3. Plot the accuracy of both fills against the offset within the run. Explain the shape of the curves by the context that each masked byte has lost, and explain where the two fills differ.

6. **Time's arrow.** ⊠ The masked loss of Section 11.4.2.3 takes nearly equal values at the two edges of the window, although position 0 sees only what follows it and position 63 only what precedes it. The conditional entropy $H(A \mid B) = -\mathbb{E}\left[\log p(A \mid B)\right]$ is the expected loss, in nats, of the best predictor of A from B.
   1. Let $X_1, X_2, \ldots$ be a stationary process, so that the joint distribution of any k consecutive variables does not depend on where the block starts. Show that
   $$H(X_1 \mid X_2, \ldots, X_{k+1}) = H(X_{k+1} \mid X_1, \ldots, X_k).$$
   2. A stationary Markov chain on the states $\{0, 1, 2\}$ moves from state i to state $i + 1 \bmod 3$ with probability 0.9 and stays with probability 0.1. Compute both sides of the identity for $k = 1$ in nats. Find the transition matrix of the time-reversed chain, and decide whether a long sample path reveals the direction of time.

3. State what the identity predicts for the two edge losses of a model whose masked-token predictions equal the true conditional distributions. Which differences between the experiment and the assumptions of the identity allow the two measured edge losses to differ?

7. code **Content-addressed alignment.** ▣ In reversal, target position t copies the same source position for every string. Change the line of `sample_batch` that builds the target so that the target is the source sorted in increasing order.
   1. For one string with a repeated letter, list the source positions that each target position may copy. Redefine the hit rate of Section 11.4.3.5 so that every correct choice counts.
   2. Train `EncoderDecoder` on sorting with one block and with two blocks, for as many steps as the exact match on fresh strings requires or until it stops improving. Report the exact match and your hit rate.
   3. Plot the maps for two strings. Describe what the decoder's query at step t must encode for sorting, compared with reversal, and which information the encoder must supply when a letter occurs more than once.
8. [extended] **A Perceiver for reversal.** ▣ Perceiver IO produces its outputs through learned output queries rather than an autoregressive decoder. Solve the reversal task of Section 11.4.3.1 in this way, with `num_hiddens=128` throughout.
   1. Embed the source with learned letter and position tables and read it with `PerceiverEncoder` into M latents. Let 12 learned output queries, one per target position, cross-attend into the latent summary, and add each result to its query. Apply `d2l.FeedForward` with a second residual connection, and map the result linearly onto the 16 letters. Train all outputs in parallel with the cross-entropy loss on fresh batches from `sample_batch`, with no decoder input.
   2. Train with $M \in \{1, 2, 4, 12\}$ for a fixed number of steps and report the exact match on 1000 fresh strings. Find the smallest M that solves the task. For a smaller M, decide whether the latents' read of the source or the output queries' read of the latents limits the model.
   3. Plot the latents' cross-attention over the source and the output queries' attention over the latents. Describe how the model divides the string among the latents.
   4. Explain why predicting all target positions independently given the latents is no restriction for reversal, and name a sequence task for which it would be one.

11.5 Vision Transformer

The convolutional networks of Chapter 6 were designed for images. They build in two assumptions (Section 6.1): *locality*, by which a unit looks only at a small neighborhood, and *translation equivariance*, by which a shifted input produces a shifted output. A transformer block assumes neither. It maps a sequence of vectors to a sequence of the same shape, and nothing in it says that the vectors stand for words, or that two of them are neighbors. Can the same blocks, told nothing about images, learn to classify them?

The first obstacle is the choice of tokens. One token per pixel is far too expensive, because attention scores every pair of tokens. Even a small 96×96 image has 9,216 pixels, so a single attention head in a single layer would score $9{,}216^2 \approx 85$ million pairs of pixels; at the 224×224 resolution of ImageNet the count exceeds 2.5 billion. Early attention models for vision therefore kept the computation local or the images small. Ramachandran et al. (2019) restricted self-attention to small windows. Cordonnier et al. (2020) proved that a self-attention layer with enough heads can express any convolution, and applied full attention

to 2×2 patches of small images.

The *vision transformer* (ViT) of Dosovitskiy et al. (2021) takes a coarser route. It cuts the image into patches of 16×16 pixels, embeds each patch as one token, and passes the sequence to an unmodified transformer encoder. Our 96×96 image becomes 36 tokens, and each head scores 1,296 pairs. Nothing in a ViT tells it that two patches are neighbors.

This section builds a ViT, trains it on Fashion-MNIST, and checks whether its position embeddings learn the layout of the patch grid. It then compares the ViT with a convolutional network of the same size, trained with the same recipe for the same number of epochs.

11.5.1 Patches as Tokens

Consider an image with height h, width w, and c channels, and a patch side p that divides both h and w. Cutting the image into a grid of non-overlapping patches gives $m = hw/p^2$ patches, and each patch flattens to a vector of length cp^2. A linear map then projects every flattened patch to a vector of width d, the width of the transformer. For our grayscale images, $h = w = 96$, $c = 1$, and $p = 16$, which gives a 6×6 grid of $m = 36$ patches of 256 pixels each.

Cutting and projecting can be done in a single operation, a convolution whose kernel size and stride both equal p. At each output position, the $c \times p \times p$ kernel of output channel k covers exactly one patch, and the output is the inner product of the kernel with that patch, which is entry k of the patch's linear projection. The convolution's weight tensor of shape (d, c, p, p) is the projection matrix, reshaped. `PatchEmbedding` applies this convolution and flattens the grid of outputs into a sequence.

```
class PatchEmbedding(nn.Module):
    """Image-to-sequence stem of the vision transformer."""
    def __init__(self, img_size=96, patch_size=16, num_hiddens=512,
                 num_channels=3):
        super().__init__()
        # A partial trailing patch would silently shrink the token grid
        assert img_size % patch_size == 0, 'patches must tile the image'
        self.num_patches = (img_size // patch_size) ** 2
        self.conv = nn.Conv2d(num_channels, num_hiddens,
                              kernel_size=patch_size, stride=patch_size)

    def forward(self, X):  # (batch, c, h, w) -> (batch, patches, d)
        return self.conv(X).flatten(2).transpose(1, 2)
```

A minibatch of four 96×96 color images, cut into 16×16 patches, should become four sequences of 36 tokens of width 512:

```
img_size, patch_size, num_hiddens, batch_size = 96, 16, 512, 4
patch_emb = PatchEmbedding(img_size, patch_size, num_hiddens)
X = torch.zeros(batch_size, 3, img_size, img_size)
d2l.check_shape(patch_emb(X),
                (batch_size, (img_size//patch_size)**2, num_hiddens))
```

From here on the image is a sequence of token vectors, and the rest of the model is the encoder of Section 11.4: a stack of transformer blocks without a causal mask, in which

every token attends to every other.

11.5.2 The Vision Transformer

Figure 11.8 shows the whole model. The patch tokens enter at the bottom, together with one extra token that belongs to no patch. The stack of blocks processes all of them, and the head reads the final representation of the extra token alone. We build the pieces in turn: the block, the extra token, and the position embeddings.

11.5.2.1 The Encoder Block

The body of the model is the pre-norm transformer block of Section 11.1. Each of its two sublayers, multi-head self-attention and a two-layer MLP, reads a normalized copy of the residual stream and adds its output back, so the stream itself is never normalized, only added to. ViT adopted this arrangement after analyses had shown that it trains deep stacks more reliably than the post-norm original (Baevski and Auli 2018; Wang et al. 2019; Xiong et al. 2020), and most transformers have used it since.

`ViTBlock` is therefore a `d2l.TransformerBlock` with two changes, both from the original ViT recipe. It normalizes with LayerNorm (`norm='layer'`), which centers and scales each token vector. And it replaces the FFN through the `ffn_factory` hook by a two-layer MLP of free width `mlp_num_hiddens` with the Gaussian error linear unit (GELU), a smooth relative of ReLU (Hendrycks and Gimpel 2016), in its exact form, and dropout after the hidden layer. Together with the dropout that `TransformerBlock` applies to the output of each sublayer, this places dropout where the original ViT does: after attention and after each of the MLP's two linear layers. Because every patch may attend to every other, the block needs no mask: the `mask` argument stays `None` throughout this section.

```
class ViTBlock(d2l.TransformerBlock):
    """Pre-norm block with LayerNorm and an exact-GELU MLP."""
    def __init__(self, num_hiddens, mlp_num_hiddens, num_heads, dropout,
                 use_bias=False):
        super().__init__(
            num_hiddens, num_heads, dropout=dropout, norm='layer',
            bias=use_bias, ffn_factory=lambda: nn.Sequential(
                nn.Linear(num_hiddens, mlp_num_hiddens),
                nn.GELU(),  # Exact GELU
                nn.Dropout(dropout),
                nn.Linear(mlp_num_hiddens, num_hiddens)))
```

As with every transformer block, the output shape equals the input shape, so the blocks can be stacked:

```
X = torch.ones(2, 100, 24)
encoder_blk = ViTBlock(24, 48, 8, 0.5)
encoder_blk.eval()
d2l.check_shape(encoder_blk(X), X.shape)
```

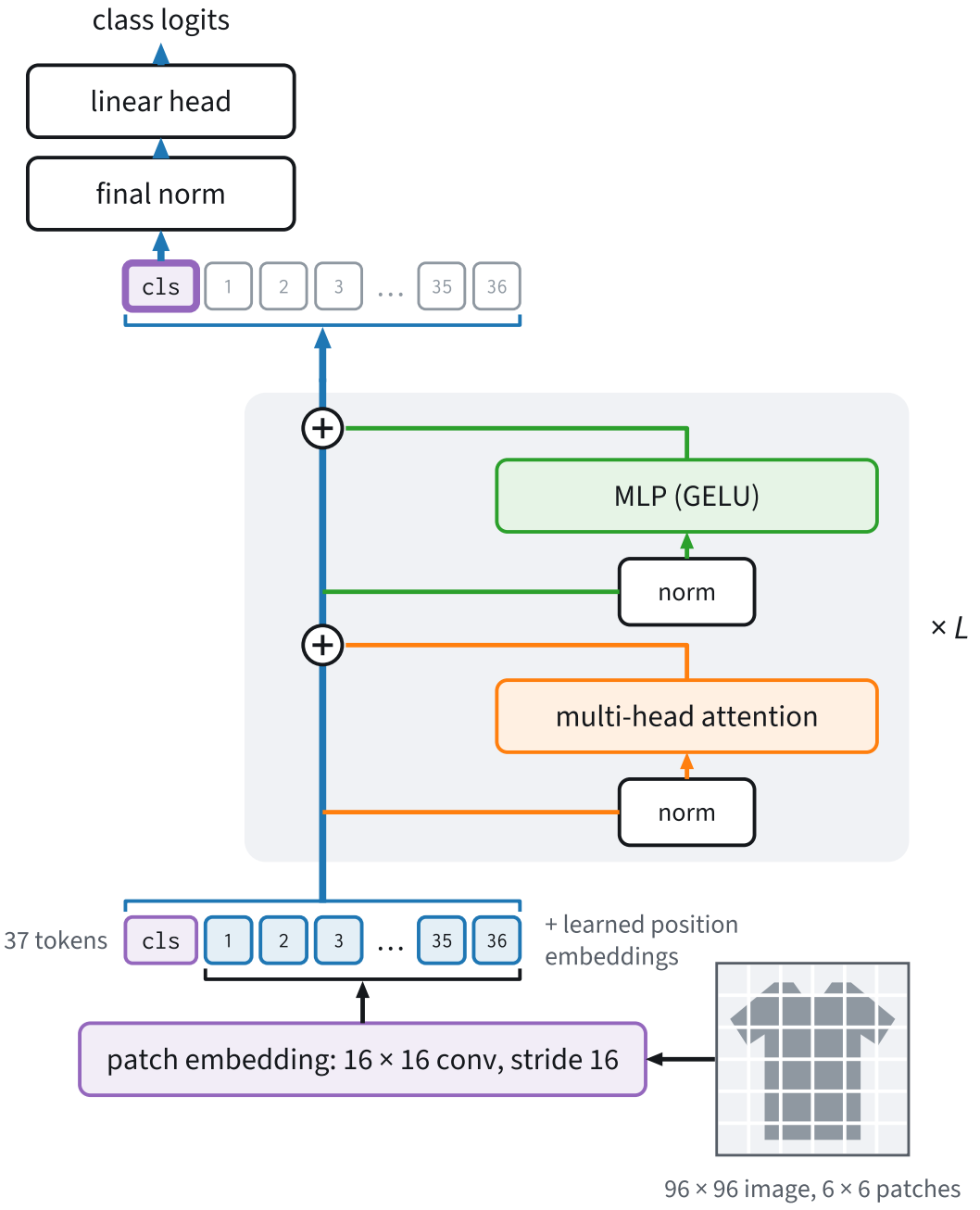


Figure 11.8: The vision transformer on a 96×96 image. A strided convolution turns the 6×6 grid of 16×16 patches into 36 tokens. A learned "<cls>" token is prepended and learned position embeddings are added; L pre-norm transformer blocks process all 37 tokens, and a linear head classifies the final representation of the "<cls>" token.

11.5.2.2 The Class Token and Position Embeddings

Two more pieces turn the stack into a classifier. The first is a learned vector, the "<cls>" (class) token, that is prepended to the 36 patch tokens and initialized to zeros. It corresponds to no patch. In every block it attends to all patches, so its final representation can collect whatever evidence the classification needs, and the head reads the class from it alone. Averaging the final representations of all patches works about as well (Dosovitskiy et al. 2021); the exercises compare the two readouts.

The second piece tells the model where each patch sits. Without position information, self-attention is permutation equivariant (Section 10.4.1): permuting the input tokens permutes the outputs in the same way, so a model that reads one output token cannot distinguish an image from any rearrangement of its patches. A patch has a row *and* a column, and both must be encoded. A fixed *2-D sinusoidal code* uses half of each vector for the 1-D sinusoidal code of the row (Section 10.4) and the other half for that of the column (Beyer et al. 2022). A 2-D form of RoPE rotates queries and keys by angles proportional to the row and to the column (Heo et al. 2024). The original ViT does neither. It learns one free vector per token position, initialized to small Gaussian noise, and adds it to the token. We follow the original recipe. At initialization the model therefore has no notion that patch 7 lies directly to the left of patch 8 and directly above patch 13; whatever spatial structure it uses, it must discover during training.

`ViT` assembles the pieces: patch embedding, class token, position embeddings with dropout, `num_blks` blocks, and a head that normalizes the class token's final representation and maps it to the logits. The class also fixes its training recipe in `configure_optimizers`, which Section 11.5.3.1 explains.

```
class ViT(d2l.Classifier):
    """Vision transformer."""
    def __init__(self, *, img_size, patch_size, num_hiddens, mlp_num_hiddens,
                 num_heads, num_blks, num_steps, emb_dropout=0.0,
                 blk_dropout=0.0, lr=1e-3, use_bias=False, num_classes=10,
                 num_channels=1):
        super().__init__()
        self.save_hyperparameters()
        self.patch_embedding = PatchEmbedding(
            img_size, patch_size, num_hiddens, num_channels)
        self.cls_token = nn.Parameter(torch.zeros(1, 1, num_hiddens))
        num_tokens = self.patch_embedding.num_patches + 1  # Add the cls token
        # Positional embeddings are learnable, initialized to small noise
        self.pos_embedding = nn.Parameter(
            torch.randn(1, num_tokens, num_hiddens) * 0.02)
        self.dropout = nn.Dropout(emb_dropout)
        self.blks = nn.ModuleList([
            ViTBlock(num_hiddens, mlp_num_hiddens, num_heads, blk_dropout,
                     use_bias=use_bias) for _ in range(num_blks)])
        self.head = nn.Sequential(nn.LayerNorm(num_hiddens, eps=1e-5),
                                  nn.Linear(num_hiddens, num_classes))

    def forward(self, X):
        X = self.patch_embedding(X)  # (batch, patches, d)
        cls = self.cls_token.expand(X.shape[0], -1, -1)
```

```
        X = self.dropout(torch.cat((cls, X), 1) + self.pos_embedding)
        for blk in self.blks:
            X = blk(X)
        return self.head(X[:, 0])

    def configure_optimizers(self):
        # AdamW decays weight matrices and kernels; it exempts biases, norm
        # parameters, the class token, and the position embeddings
        named = list(self.named_parameters())
        decay = lambda n, p: p.ndim >= 2 and n not in ('cls_token',
                                                       'pos_embedding')
        optim = torch.optim.AdamW(
            [{'params': [p for n, p in named if decay(n, p)]},
             {'params': [p for n, p in named if not decay(n, p)],
              'weight_decay': 0.0}], lr=self.lr, weight_decay=0.05)
        # Linear warmup over the first tenth of the steps, then cosine
        return optim, torch.optim.lr_scheduler.LambdaLR(optim, lambda t: (
            d2l.cosine_warmup(t, self.num_steps, self.num_steps // 10)))
```

11.5.2.3 Register Tokens

The class token is an input with no image content, a place where the model can collect and store information. Large ViTs turn out to need more such places. Darcet et al. (2024) found that ViTs of the size of ViT-Large and above, once trained long enough, take over about 2% of the patch tokens, mostly in uninformative background regions, and use them for internal computation. The outputs of these tokens have norms about ten times those of the others, they carry little information about their own patch, and they show up as bright spots in the attention maps. This is the vision counterpart of the attention sinks of Section 11.3.4.2, positions that receive a large share of the attention whether or not their content is relevant to the query.

The remedy is to provide the storage explicitly. A few additional learned tokens, called *registers*, are appended to the input like the class token and discarded at the output. One register already removes the artifacts, and four, the number used in most of the paper's experiments, improve the tasks that read the patch outputs, such as segmentation, depth estimation, and object discovery, while adding less than 2% to the computation. Our two-block model is far below the size at which the artifacts appear, so we train it without registers.

11.5.3 Training on Fashion-MNIST

11.5.3.1 A First Run

We train a small ViT on Fashion-MNIST, with the images resized to 96×96 so that 16×16 patches give the 6×6 grid of 36 tokens. The model has two blocks of width 512 with eight heads and an MLP of width 2,048, about 6.5 million parameters in all.

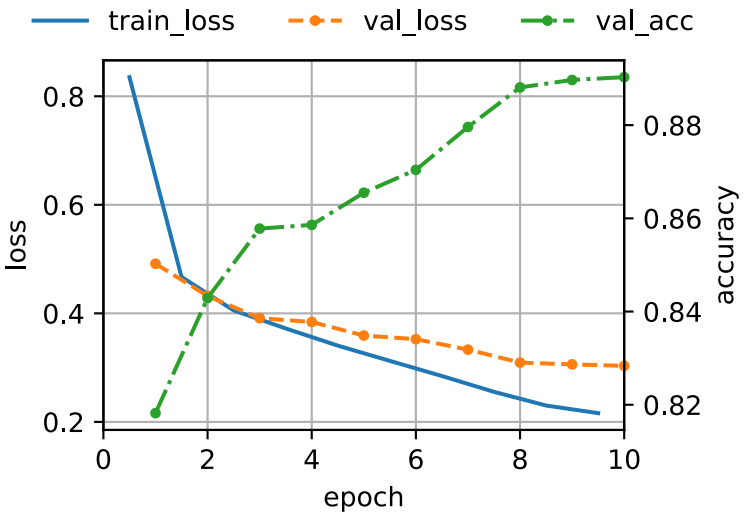
The training recipe matters more for a transformer than for a CNN. In the ablation of Touvron, Cord, Douze, et al. (2021), replacing AdamW by SGD, with everything else unchanged, lowered the ImageNet accuracy of a ViT-Base from 81.8% to 74.5%. `ViT` therefore uses the optimizer and schedule of the compact modern recipe of Section 7.6: AdamW with weight decay 0.05, and a learning rate that rises linearly from zero and then decays to zero along a half-cosine Equation 7.1. We tuned each model's base rate over $\{3 \times 10^{-4}, 10^{-3}, 3 \times 10^{-3}\}$.

The ViT trains best at 3×10^{-4} and ends four to five points lower at 10^{-3}; the CNN of Section 11.5.4.1 varies by about a point over the three rates and does equally well at 10^{-3} and 3×10^{-3}, so we use 10^{-3} for it. For both models the warmup takes the first tenth of the minibatch steps, which over ten epochs is the first epoch. The decay acts on the weight matrices and convolution kernels only. Biases and normalization parameters are exempt, as in Section 9.5.4.5, and so are the class token and the position embeddings, as in the training code of Touvron, Cord, Douze, et al. (2021). Decaying the position embeddings would also pull them toward zero while we try to measure what they learn. We leave out the recipe's label smoothing and Mixup, regularizers that help over long runs, so that the ViT and the CNN it is compared with later both minimize the plain cross-entropy for ten epochs.

```
data = d2l.FashionMNIST(batch_size=128, resize=(96, 96))
num_steps = 10 * len(data.train_dataloader())  # Ten epochs of minibatches
hparams = dict(img_size=96, patch_size=16, num_hiddens=512,
               mlp_num_hiddens=2048, num_heads=8, num_blks=2,
               emb_dropout=0.1, blk_dropout=0.1, num_steps=num_steps)
```

```
model = ViT(**hparams, lr=3e-4)
```

```
trainer = d2l.Trainer(max_epochs=10, num_gpus=1)
trainer.fit(model, data)
vit_history = trainer.history
vit_params, vit_acc = d2l.num_params(model), vit_history['val_acc'][-1]
print(f'ViT: {vit_params/1e6:.1f}M parameters, '
      f'validation accuracy {vit_acc:.2f}')
```




```
ViT: 6.5M parameters, validation accuracy 0.89
```

The ViT reaches a validation accuracy of about 90%. Before we ask whether that is good for a model of its size, we look at what it has learned about the position of its patches.

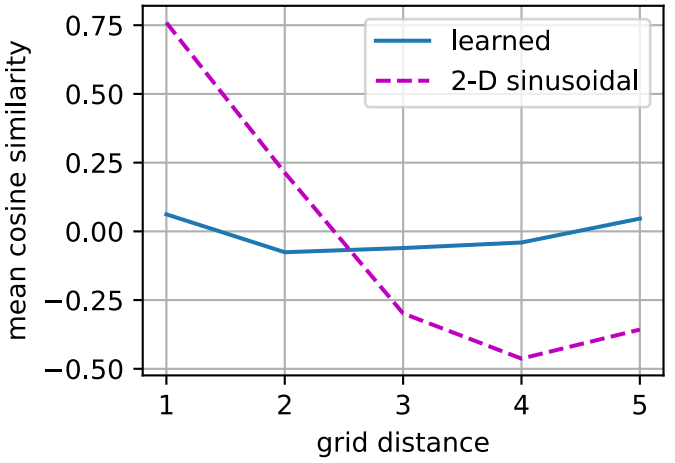
11.5.3.2 Grid Structure in the Learned Embeddings

The position embeddings started as noise, and nothing in the training signal ever names a row or a column. In fully trained ViTs they reflect the grid nonetheless. In the analysis of Dosovitskiy et al. (2021), each position's embedding is most similar to those of its neighbors and of its own row and column. Does our small model show this after ten epochs?

We measure the structure with one curve. For every pair of the 36 patch positions we compute the cosine similarity of their embeddings, after subtracting the mean embedding over all positions: a component shared by every position says nothing about where a patch is. We then average the similarities over all pairs at the same grid distance, counted as the larger of the row offset and the column offset, so that the eight cells around a patch are at distance 1. Embeddings that encode the grid give a curve that falls with distance. Independent random vectors in 512 dimensions give a flat curve slightly below zero, near $-1/35$: once centered, the 36 vectors sum to zero, which makes their average pairwise cosine negative. As a reference we compute the same curve for the fixed 2-D sinusoidal code, which encodes the grid by design.

```
def grid_similarity(P):
    """Mean cosine similarity of 36 position embeddings, by grid distance."""
    P = P - P.mean(0)                    # Drop the part shared by all
    P = P / torch.linalg.vector_norm(P, dim=1, keepdim=True)
    sim = P @ P.T                        # (36, 36) cosine similarities
    rows, cols = torch.meshgrid(torch.arange(6), torch.arange(6),
                                indexing='ij')
    ij = torch.stack((rows.flatten(), cols.flatten()), 1)  # Grid cells
    dist = (ij[:, None] - ij[None]).abs().amax(-1)  # Max of the two offsets
    return [float(sim[dist == k].mean()) for k in range(1, 6)]

S = d2l.sinusoidal_encoding(6, 256)      # 1-D codes of the 6 rows/columns
rows, cols = torch.meshgrid(torch.arange(6), torch.arange(6), indexing='ij')
fixed = torch.cat((S[rows.flatten()], S[cols.flatten()]), 1)  # (36, 512)
learned = model.pos_embedding[0, 1:].detach().cpu()  # Drop the cls slot
d2l.plot(list(range(1, 6)), [grid_similarity(learned), grid_similarity(fixed)],
         'grid distance', 'mean cosine similarity',
         legend=['learned', '2-D sinusoidal'])
```



The fixed code's curve falls from about 0.76 for neighboring patches to between -0.3 and -0.5 at distances 3 to 5, because by construction patches in nearby rows and columns get similar codes. The learned embeddings show only a trace of this structure. Their curve stays close to zero: neighbors are slightly more similar than patches three cells apart, by about a tenth, and the curve rises again at distance 5, where every pair joins two opposite edges of the image. Fashion-MNIST items rarely reach the edges, so the border patches mostly see background, which may make their embeddings alike; an exercise tests whether the small dip is more than chance. Ten epochs on 60,000 images teach the ViT little about its own geometry. The network that we compare with next is given this geometry by its architecture.

11.5.4 Comparison with a Convolutional Network

11.5.4.1 A Convolutional Baseline at the Same Budget

Is 90% good for a model of 6.5 million parameters trained for ten epochs? We compare the ViT with a CNN built from the residual blocks of Section 7.4, each of which adds the output of two 3×3 convolutions with batch normalization to its input. `CompactResNet` uses the familiar 7×7 stem and then seven residual blocks whose widths bring the total to the ViT's parameter count. Everything else is held equal. Both models see the same data for ten epochs, and both use the same recipe: `CompactResNet` borrows `configure_optimizers` from `ViT`. The two also do about the same work, about half a billion floating-point operations per image in the forward pass (the exercises derive both counts).

```
class CompactResNet(d2l.Classifier):
    """A ResNet-style CNN parameter-matched to the ViT above."""
    configure_optimizers = ViT.configure_optimizers  # The same recipe

    def __init__(self, num_steps, lr=1e-3, num_classes=10):
        super().__init__()
        self.save_hyperparameters()
        self.net = nn.Sequential(
            nn.LazyConv2d(64, kernel_size=7, stride=2, padding=3, bias=False),
            nn.LazyBatchNorm2d(), nn.ReLU(),
```

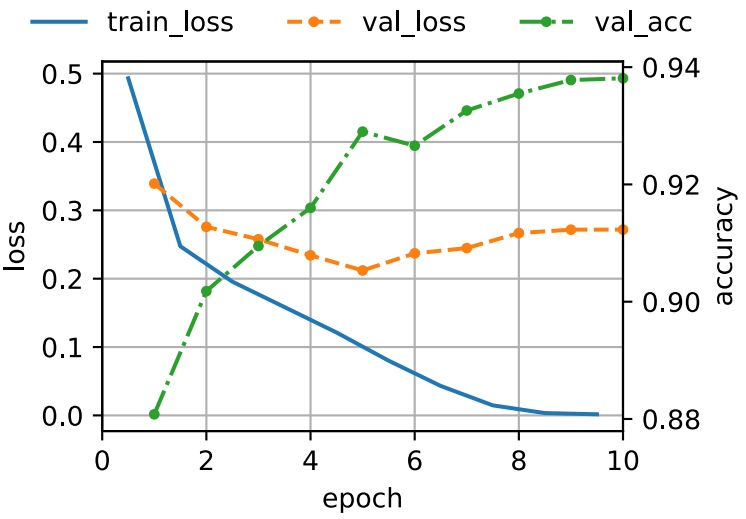
```
            nn.MaxPool2d(kernel_size=3, stride=2, padding=1),
            d2l.Residual(64), d2l.Residual(64),
            d2l.Residual(128, strides=2), d2l.Residual(128),
            d2l.Residual(256, strides=2), d2l.Residual(256),
            d2l.Residual(512, strides=2),
            nn.AdaptiveAvgPool2d((1, 1)), nn.Flatten(),
            nn.LazyLinear(num_classes))

    def forward(self, X):
        return self.net(X)

    def weight_init(self, module):
        # The Trainer calls this after the first minibatch fixed the shapes
        if type(module) in (nn.Linear, nn.Conv2d):
            nn.init.kaiming_normal_(module.weight, nonlinearity='relu')  # He
            if module.bias is not None:
                nn.init.zeros_(module.bias)
```

```
cnn = CompactResNet(num_steps, lr=1e-3)
```

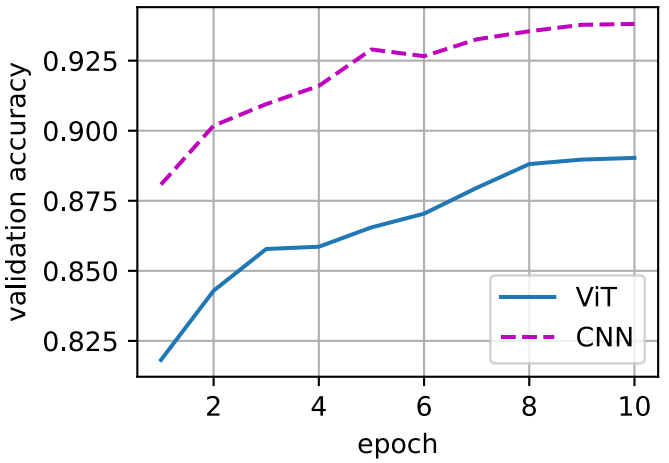
```
trainer = d2l.Trainer(max_epochs=10, num_gpus=1)
trainer.fit(cnn, data)
cnn_history = trainer.history
cnn_params, cnn_acc = d2l.num_params(cnn), cnn_history['val_acc'][-1]
print(f'CNN: {cnn_params/1e6:.1f}M parameters, '
      f'validation accuracy {cnn_acc:.2f}')
print(f'ViT: {vit_params/1e6:.1f}M parameters, '
      f'validation accuracy {vit_acc:.2f}')
```




```
CNN: 6.5M parameters, validation accuracy 0.94
ViT: 6.5M parameters, validation accuracy 0.89
```

Plotting the validation accuracy of the two runs against the epoch shows how the gap develops over training:

```
d2l.plot(list(range(1, 11)), [vit_history['val_acc'], cnn_history['val_acc']],
         'epoch', 'validation accuracy', legend=['ViT', 'CNN'])
```



At the same parameter count, data, recipe, and number of epochs, and at about the same computational cost, the CNN reaches about 94%, four to five points ahead of the ViT. The experiment measures the two models under one protocol; it does not say which property of the CNN produces the gap. The CNN builds in locality and translation equivariance, which the ViT must infer from 60,000 images, and the position embeddings of Section 11.5.3.2 show how much of the spatial layout the ViT has inferred after ten epochs. But the two models also differ in depth, in normalization, and in how finely they look at the image.

11.5.4.2 The Comparison at Other Scales

The patch size alone moves the result by two points on Fashion-MNIST and by sixteen on CIFAR-10. Training for 300 epochs with heavy augmentation, Hassani et al. (2021) report 93.2% on Fashion-MNIST for ViTs of about 3.7 to 3.9 million parameters with 16×16 patches, 94.5% with 8×8 patches, and 95.2% with 4×4 patches, slightly above the 94.8% of a ResNet-18. On CIFAR-10 the same three models span 78% to 94%, and only the smallest patches beat the ResNet-18. Smaller patches mean more tokens, so the ViT sees finer detail at a higher cost: halving p quadruples the number of tokens and roughly quadruples the computation, while the parameter count barely changes.

The ranking also changes with the amount of data. When Dosovitskiy et al. (2021) pretrained on the 300 million images of JFT-300M and fine-tuned on smaller benchmarks, their largest ViTs exceeded the convolutional baselines they compared with. In a controlled version, the same models were pretrained on random subsets of JFT-300M of 9 million to 300 million images: the ResNets led on the smaller subsets, and the ViTs overtook them on the larger ones. Touvron, Cord, Douze, et al. (2021) then trained ViTs on the 1.3 million images of ImageNet alone, with strong augmentation and long training, and made them competitive with convolutional networks of similar cost. With enough data or a strong enough recipe, the assumptions that the CNN builds in are no longer needed for accuracy, and

the transformer contributes its generality: the same block serves text, images, and both together.

11.5.5 Summary

A vision transformer turns an image into a sequence by cutting it into patches and embedding each patch with one strided convolution; an unmasked encoder stack then processes the patch tokens, and a linear head classifies the final representation of a learned class token. The architecture is the text transformer's, and only the tokenization and the readout are new. Position embeddings must encode a row and a column. Learned from noise, as in the original ViT, they show only a trace of the grid after ten epochs on Fashion-MNIST, while fixed 2-D sinusoidal codes build it in. Large ViTs also need a few register tokens as internal storage. Matched with a CNN in parameters, computation, data, and training recipe, our ViT trails by four to five points; smaller patches and much more data narrow or reverse such gaps.

11.5.6 Exercises

1. code **Parameters versus compute.** The ViT of Section 11.5.3.1 reads grayscale images of $h = w = 96$ pixels with $c = 1$ channel through patches of side $p = 16$. It has width $d = 512$, MLP width $4d$, and two blocks. Count one multiply–add as two FLOPs, as Equation 10.16 does, and write $n = m + 1$ for the number of tokens.
    1. Express the parameter counts of `PatchEmbedding` and of the position embeddings as functions of p. By how many parameters does the model change when p is halved to 8, and what fraction of its 6.5 million parameters is this change?
    2. Show that the stem costs the same number of FLOPs per image for every p. Derive the FLOPs of one `ViTBlock` as a function of n and d, and compute the factor by which the forward FLOPs of the model grow when p is halved to 8.
    3. Find the number of tokens at which the term of your block count that is quadratic in n equals the remaining terms. Which patch sizes put this model beyond that point, and how far below it does ViT-B/16 operate, with $d = 768$ at a resolution of 224×224?
    4. `CompactResNet` was matched to the ViT by parameter count. Count its convolution FLOPs per image, ignoring normalization, activations, and pooling, and compare them with those of the ViT.
    5. Train the ViT with $p = 8$, with the protocol of Section 11.5.3.1, and compare its validation accuracy with `vit_acc` and `cnn_acc`. Which of the two comparisons with the CNN, at equal parameters or at equal computation, does this run inform?
2. code **Choice of readout.** `ViT` classifies from the final representation of the "<cls>" token.
    1. Under this readout, part of the computation of the last `ViTBlock` cannot influence the logits. Identify it, and compute the fraction of the FLOPs of that block that inference could skip in the configuration of Section 11.5.3.1, counting as in problem 1.
    2. Implement the alternative readout that averages the final representations of the 36 patches, and remove the "<cls>" token together with its position embedding. Train the model with the protocol of Section 11.5.3.1, and compare its validation accuracy with `vit_acc`.

3. Dosovitskiy et al. (2021) report that the two readouts work about equally well but need different learning rates. Train both readouts at the learning rates 10^{-4} and 10^{-3} as well. Does the ranking of the two readouts depend on the learning rate?

3. code **Position information.** By the proposition of Section 10.4.1, self-attention without positional information is equivariant to permutations of its input tokens.

   1. Suppose that the 36 patch position embeddings of a `ViT` are all equal. Show that its logits are then unchanged when the patch tokens are permuted, and that permuting the 16×16 pixel blocks of the input image has the same effect as permuting the tokens.
   2. Write a variant of `ViT.forward` that applies a fixed permutation to the tokens produced by `PatchEmbedding`, before the position embeddings are added. Measure the validation accuracy of the trained `model` under five random permutations.
   3. Train a `ViT` whose position embeddings are fixed at zero, with the protocol of Section 11.5.3.1. Compare its validation accuracy with `vit_acc`, and confirm that the five permutations leave its predictions unchanged.
   4. Train a `ViT` whose patch position embeddings are fixed to the 2-D sinusoidal code `fixed` of Section 11.5.3.2, with a learned embedding for the "<cls>" slot. Compare its validation accuracy with `vit_acc` and with the model of the previous part.
   5. Compare the accuracy drop under permutation with the gaps between the three trained models. What does each of these numbers measure?

4. code **Grid permutation test.** Let Δ be the difference between the first and the third value that `grid_similarity` returns for the learned embeddings, the mean similarity at grid distance 1 minus that at grid distance 3.

   1. Permute the rows of `learned` at random, which reassigns the embeddings to grid cells, recompute Δ, and repeat 1,000 times. In what fraction of the permutations is Δ at least as large as for the trained `model`? What does this fraction say about the evidence for grid structure after ten epochs?
   2. Train a `ViT` with the protocol of Section 11.5.3.1 for 30 epochs instead of 10, with `num_steps` set to 30 epochs of minibatches, and repeat the test. How do Δ and its position within the permutation distribution change with longer training?

5. **Shift equivariance.** Shift a 96×96 image by s pixels to the right, and fill the vacated columns with zeros. Suppose that the s rightmost columns of the original image are zero.

   1. For $s = 16$, describe the tokens that `PatchEmbedding` produces for the shifted image, and show that they are a rearrangement of those of the original image. Give an image for which this fails at $s = 8$.
   2. Use this and problem 3 to show that a `ViT` whose patch position embeddings are all equal assigns the same logits to both images when $s = 16$. Why does the argument not extend to the trained `model`?
   3. Compute the stride of the final feature map of `CompactResNet` relative to its input. Show that, for cells whose receptive field lies inside the image, a shift of the input by a multiple of this stride moves the final feature map by whole cells, and explain why no such statement holds for a shift by fewer pixels. Compute the side length of the receptive field of a cell of the final feature map with Equation 6.12. Does any cell of the final feature map of a 96×96 input meet the condition?
   4. Neither model is therefore built to be invariant to a shift by one pixel. Name a property that every convolution of `CompactResNet` builds in and the attention layers

of the ViT lack, and a property that both models share.

6. code **Attention distance.** ▯ For one head, let α_{ij} be the attention weight of query patch i on key patch j in an image, and let $\mathbf{u}_i$ be the pixel coordinates of the center of patch i. The *mean attention distance* of the head (Dosovitskiy et al. 2021) averages

$$\bar{r} = \frac{1}{36} \sum_{i=1}^{36} \sum_{j=1}^{36} \frac{\alpha_{ij}}{\sum_{k=1}^{36} \alpha_{ik}} \|\mathbf{u}_i - \mathbf{u}_j\| \tag{11.13}$$

over images, with the "<cls>" token excluded as query and key. In large pretrained ViTs, some heads of the lowest layers attend locally while others attend across the whole image, and the distance grows with depth.

   1. Evaluate $\bar{r}$ for a head that attends uniformly to all 36 patches, as a reference value.
   2. After a forward pass of the trained `model` in evaluation mode, `blk.attention.attention_weights` holds the attention weights of block `blk`, of shape (batch size, 8, 37, 37). Compute $\bar{r}$ for each of the 16 heads on 512 validation images, and plot it against the block index. Do the two blocks show either pattern of the pretrained ViTs?
   3. Repeat the measurement for the ViT of problem 3 whose position embeddings are fixed at zero. What other than position embeddings can make a head attend to nearby patches, and how do the results of the two models separate the two causes?

7. **Shifted windows.** ▯ Swin transformers (Liu et al. 2021) restrict attention to non-overlapping windows of $M \times M$ patches. Successive blocks alternate between a regular partition of the patch grid into windows and a partition displaced by $\lfloor M/2 \rfloor$ patches along both axes. Normalization and MLPs act on each token separately. Call the set of input patches that can influence the output of a token after L blocks its *reach*. A window is the product of a row interval and a column interval, so the reach is the product of its one-dimensional counterparts, and it suffices to study one row of patches far from the border of the grid.

   1. Show that without the displaced partition, the reach of every token remains its own window at every depth.
   2. Show that with the alternation, the reach after L blocks is an interval of ML patches. Compare this growth with that of a stack of 3×3 convolutions of stride 1 and with a single layer of global attention. On a 56×56 grid with $M = 7$, after how many blocks can the token in row 28 and column 28, counting from 0, be influenced by every patch?
   3. Show that the length ML stays the same for every displacement s with $0 < s < M$. After two blocks, let $g(s)$ be the smallest distance from a token to the nearer end of its reach, taken over all tokens. Compute $g(s)$, and show that $s = \lfloor M/2 \rfloor$ maximizes it.

8. [extended] **Distillation from the CNN.** ▯▯ Knowledge distillation trains a student model to match the predictions of a trained teacher model as well as the labels (Hinton et al. 2015). Touvron, Cord, Douze, et al. (2021) find that ViTs distilled from a convolutional teacher outperform ViTs distilled from a transformer teacher, and suggest that the student inherits the inductive bias of the teacher. Use the trained `cnn` as the teacher and a fresh `ViT` as the student. Let $\mathbf{z}_s$ and $\mathbf{z}_t$ be the logits of student and

teacher for an image with label y, let CE be the cross-entropy of the softmax of its first argument, and let y_t be the class with the largest teacher logit. Hard-label distillation minimizes

$$\mathcal{L}_{\text{hard}} = \tfrac{1}{2}\,\text{CE}(\mathbf{z}_s, y) + \tfrac{1}{2}\,\text{CE}(\mathbf{z}_s, y_t), \tag{11.14}$$

and soft distillation at temperature τ and weight λ minimizes

$$\mathcal{L}_{\text{soft}} = (1-\lambda)\,\text{CE}(\mathbf{z}_s, y) + \lambda\tau^2\,\text{KL}\big(\text{softmax}(\mathbf{z}_t/\tau)\,\|\,\text{softmax}(\mathbf{z}_s/\tau)\big). \tag{11.15}$$

1. Compute the teacher's logits on all training images once, and deliver them with each minibatch, so that the training step of the student needs no forward pass of the teacher. On what fraction of the training images is $y_t \neq y$? Explain what this fraction implies for the difference between $\mathcal{L}_{\text{hard}}$ and plain cross-entropy in this setting, and why the implication does not carry over to the augmented training of
   Touvron, Cord, Douze, et al. (2021).
2. Override `training_step` of `ViT` to minimize $\mathcal{L}_{\text{hard}}$, and train with the protocol of Section 11.5.3.1. Train again with $\mathcal{L}_{\text{soft}}$ at $\tau = 3$ for $\lambda \in \{0.1, 0.5\}$, and compare the three students with `vit_acc`.
3. For the undistilled `model` and for each student, compute the fraction of validation images on which its prediction differs from the teacher's, as in the disagreement analysis of
   Touvron, Cord, Douze, et al. (2021). Among the validation images that it misclassifies, compute also the fraction on which it predicts the same wrong class as the teacher. Which of the two numbers separates resemblance to the teacher from accuracy?
4. The teacher is both more accurate than the student and convolutional, so a gain from distillation does not by itself show that an inductive bias was transferred. Construct a control that separates the two explanations, run it, and state which outcome would favor the inductive-bias explanation.

11.6 Mixture of Experts

Every transformer of this chapter so far is *dense*: each token passes through every parameter. A forward pass costs about two floating-point operations per parameter and token (Section 11.2), so a model with twice the parameters, which can store more of what it reads, also costs twice as much to train and to run. Yet most of what a large model knows is of no use for any one token. Continuing "the integral of" draws on different knowledge from continuing "Act I, Scene". Can a model hold more parameters without spending more computation on each token?

A *mixture of experts* (MoE) does this by running only part of the model for each token. It keeps E copies of each block's feed-forward network, the *experts*, and a small learned *router* sends each token to only k of them. The stored parameters then grow with E, while the computation per token grows only with k. DeepSeek-V3, for example, stores 671 billion parameters and uses 37 billion of them for each token (DeepSeek-AI, Liu, et al. 2024).

The idea is old. Jacobs et al. (1991) trained a committee of specialist networks under a gating network, and Shazeer et al. (2017) made the gate sparse, so that only a few of thousands of

experts run for each input. This section counts stored and active parameters, implements the layer, and studies how it fails: a router left to itself sends most tokens to a few experts. We compare two remedies, an auxiliary loss and a controller that adds no gradient. Finally, we give the GPT of Section 11.2 eight experts per block and train it against the dense model at equal computation per token.

11.6.1 Conditional Computation

The feed-forward network is the natural place for experts. A block of width d spends $4d^2$ parameters on the four attention projections and, with the SwiGLU network of Section 11.1 (three matrices of size $d \times 8d/3$), $8d^2$ on the feed-forward network: two thirds of the block. And while attention mixes information between positions, the feed-forward network transforms each position on its own, so one token can be handed to a different network than its neighbor without affecting it.

Replacing the network by E experts, of which each token uses k, multiplies the stored feed-forward parameters by E and the active ones by k, and adds a router, a $d \times E$ matrix per layer. At about two operations per active parameter and token, the cost of a token follows the active count. The function below counts both for a stack of expert layers. We apply it to two deployed models and to our own configuration, the one trained at the end of this section. Mixtral 8x7B has 8 full-width experts per layer, of which each token uses 2 (Jiang et al. 2024). DeepSeek-V3 has 256 narrow experts per layer, of which each token uses 8, plus one *shared* expert that every token uses (DeepSeek-AI, Liu, et al. 2024).

```
def moe_accounting(name, d, width, num_experts, num_active, num_layers,
                   shared=0, unit=1e9):
    """Expert parameters stored, and expert and router parameters active
    per token, in a stack of SwiGLU expert layers."""
    per_expert = 3 * d * width          # SwiGLU: three d x width matrices
    store = num_layers * (num_experts + shared) * per_expert
    active = num_layers * ((num_active + shared) * per_expert
                           + d * num_experts)      # + the router
    tag = 'B' if unit == 1e9 else 'M'
    print(f'{name:>12}: {store/unit:6.1f}{tag} in experts, '
          f'{active/unit:5.1f}{tag} active per token, '
          f'ratio {store/active:4.1f}x')
    return store, active

moe_accounting('ours', 256, 683, 8, 1, 6, unit=1e6)
moe_accounting('DeepSeek-V3', 7168, 2048, 256, 8, 58, shared=1)
store, active = moe_accounting('Mixtral 8x7B', 4096, 14336, 8, 2, 32)
attn = 32 * (2 * 4096 * 4096 + 2 * 4096 * 1024)  # GQA: 32 Q, 8 KV heads
emb = 2 * 32000 * 4096                           # embedding + output head
print(f'{"":>12}  with attention and embeddings: '
      f'{(store + attn + emb)/1e9:.1f}B stored, '
      f'{(active + attn + emb)/1e9:.1f}B active')
```

```
        ours:   25.2M in experts,   3.2M active per token, ratio  8.0x
 DeepSeek-V3:  656.5B in experts,  23.1B active per token, ratio 28.4x
Mixtral 8x7B:   45.1B in experts,  11.3B active per token, ratio  4.0x
              with attention and embeddings: 46.7B stored, 12.9B active
```

DeepSeek-V3's 58 expert layers store 656B of its 671B parameters, and a token uses 23B of

them: the layers store 28 times what a token uses. (The 37B on its model card add attention, the embeddings, and three dense layers, which our count of the expert layers leaves out.) For Mixtral, adding the attention layers, which share each key and value head among four query heads (Section 11.3), and the separate embedding and output matrices gives 46.7B stored and 12.9B active parameters. The name "8x7B" thus describes what the model stores, not the work per token: a token uses less than two sevenths of the parameters. Our configuration has Mixtral's eight experts per layer but sends each token to one of them, so it stores eight times what it uses.

The count rests on two assumptions. First, it counts only computation. All E experts must sit in accelerator memory, and during training each of them receives gradients from the tokens routed to it. Large models therefore spread their experts over many devices and send every token to its experts and back, a systems problem that the Language Models part takes up. Second, the count assumes that the router spreads tokens evenly over the experts. If it sends most tokens to a few of them, the other experts hold parameters that hardly ever run, and the model behaves like a smaller one.

11.6.2 A Mixture-of-Experts Layer

The layer replaces the feed-forward network of a block, and nothing else: attention, normalization, and the residual stream stay as they are. We first define what it computes and then how to compute it efficiently.

11.6.2.1 Top-k Routing

A linear router scores the E experts for a token $\mathbf{x} \in \mathbb{R}^d$, the k largest scores select experts, and the outputs of the selected experts are added, each weighted by the router's probability:

$$\mathrm{MoE}(\mathbf{x}) = \sum_{i \in \mathcal{E}_k(\mathbf{x})} p_i(\mathbf{x})\, \mathrm{FFN}_i(\mathbf{x}), \qquad \mathbf{p}(\mathbf{x}) = \mathrm{softmax}(\mathbf{W}_r \mathbf{x}), \tag{11.16}$$

where $\mathbf{W}_r \in \mathbb{R}^{E \times d}$ is the router and $\mathcal{E}_k(\mathbf{x})$ holds the indices of the k largest entries of $\mathbf{p}(\mathbf{x})$. The layer is thus a feed-forward network chosen per token: two tokens of the same sentence may use entirely different parameters. Because each token chooses its experts without regard to the others, this is called *token-choice* routing. Figure 11.9 follows one token through a layer with $E = 8$ and $k = 2$.

The weight of a selected expert is its probability p_i itself, not a probability renormalized over the k selected experts. This choice follows the Switch transformer (Fedus et al. 2022). Selecting the top k is a discrete decision and has no gradient. The weight of a selected expert has one: if a larger contribution of expert i would lower the loss, the gradient raises p_i, and with it the chance that the router selects expert i for similar tokens. Through the softmax, every logit receives a gradient, including those of the experts that were not selected. Mixtral instead renormalizes the weights of the selected experts to sum to one (Jiang et al. 2024). The router then learns only how to divide the weight among the selected experts, and for top-1 routing, where the single weight is always one, it learns nothing from the loss at all. With raw probabilities, a top-1 router still learns. The first exercise works out these gradients.

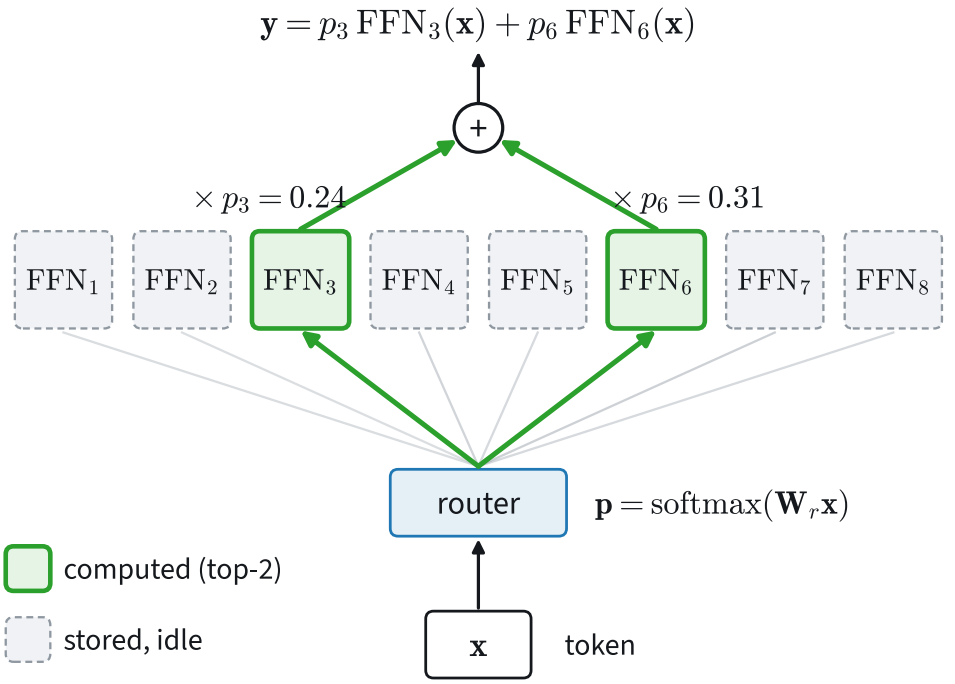


Figure 11.9: A mixture-of-experts layer with eight experts, two of which are active per token. The router scores every expert. The two most probable experts run, and their outputs are weighted by their probabilities and added. The other six experts are stored but do no work for this token.

11.6.2.2 Dispatching Tokens to Experts

Evaluating Equation 11.16 literally, by running all E experts on every token and multiplying the outputs of the unselected ones by zero, gives the right answer at E times the cost and saves nothing. An efficient layer brings each token to its experts instead. A minibatch of n tokens makes nk assignments, pairs of a token and one of its experts, and how many of them each expert receives changes from batch to batch. Matrix multiplications need fixed shapes, so each expert gets a buffer of $C = \lceil c\,nk/E \rceil$ slots, where the *capacity factor* c leaves room for an uneven spread; the Switch transformer trains with $c = 1.25$ (Fedus et al. 2022).

We number the assignments of each expert in the order of the tokens and copy the token of an expert's r^{th} assignment into slot r of that expert's buffer. An assignment with $r \geq C$ is *dropped*: that expert skips the token, which keeps the outputs of its other experts and, through the residual connection, its input. All E experts then run at once, as one batched matrix multiplication over buffers of shape (E, C, d), and each token collects its k outputs, weighted by their probabilities. The work is that of $EC \approx cnk$ rows, a fraction ck/E of running every expert on every token. Fixed shapes come with dropping: a balanced router loses few assignments, an unbalanced one many.

Besides its output, the layer keeps four things for the experiments below, which its method `record` updates for every batch: `expert_bias`, a vector added to the probabilities for the top-k selection only, which stays zero until we need it; `load`, the number of assignments each expert has received, dropped or not; `dropped`, the number of dropped assignments; and `aux_loss`, a measure of imbalance that we derive when we use it.

```
class MoELayer(nn.Module):
    """Token-choice top-k mixture of SwiGLU d2l.FeedForward experts, each
    with a buffer of C = capacity_factor * n k / E slots for n tokens."""
    def __init__(self, num_hiddens, num_experts, num_active,
                 capacity_factor=1.25):
        super().__init__()
        self.num_experts, self.num_active = num_experts, num_active
        self.capacity_factor = capacity_factor
        self.router = nn.Linear(num_hiddens, num_experts, bias=False)
        self.experts = nn.ModuleList([d2l.FeedForward(num_hiddens)
                                      for _ in range(num_experts)])
        self.register_buffer('expert_bias', torch.zeros(num_experts))
        self.register_buffer('load', torch.zeros(num_experts))
        self.register_buffer('dropped', torch.zeros(()))

    def forward(self, X):
        x = X.reshape(-1, X.shape[-1])                # (n, d): the tokens
        (n, d), E, k = x.shape, self.num_experts, self.num_active
        C = math.ceil(self.capacity_factor * n * k / E)  # slots per expert
        P = F.softmax(self.router(x), -1)             # (n, E)
        idx = (P + self.expert_bias).topk(k).indices  # (n, k)
        e = idx.flatten()                             # expert of each pair
        onehot = F.one_hot(e, E)                      # (n k, E)
        rank = (onehot.cumsum(0) * onehot).sum(1) - 1  # place in its queue
        slot = torch.where(rank < C, e * C + rank, E * C)  # E C: dropped
        buf = x.new_zeros(E * C + 1, d)               # + an overflow row
        buf[slot] = x.repeat_interleave(k, 0)
```

```
        Y = self.run_experts(buf[:-1].reshape(E, C, d)).reshape(E * C, d)
        Y = torch.cat([Y, Y.new_zeros(1, d)])[slot]    # dropped: zeros
        weights = P.gather(-1, idx)                    # (n, k): the p_i
        out = (weights[..., None] * Y.reshape(n, k, d)).sum(1)
        self.record(onehot.sum(0), (rank >= C).sum(), P)
        return out.reshape(X.shape)

    def run_experts(self, H):
        """Each expert's SwiGLU on its own buffer: (E, C, d) -> (E, C, d)."""
        W = lambda name: torch.stack(  # (E, d_in, d_out): one per expert
            [getattr(e, name).weight for e in self.experts]).mT
        return (F.silu(H @ W('W_g')) * (H @ W('W_1'))) @ W('W_2')

    def record(self, counts, num_dropped, P):
        """Add a batch's assignments per expert and drops to the totals,
        and keep its balancing loss."""
        self.load += counts
        self.dropped += num_dropped
        f = counts / counts.sum()                      # this batch's shares
        self.aux_loss = self.num_experts * (f * P.mean(0)).sum()

    def reset_stats(self):
        self.load.zero_()
        self.dropped.zero_()
```

The cumulative sum over the one-hot encoding numbers the assignments of each expert: row j counts, for every expert, the assignments to it among the first j, so its entry at the expert of assignment j is that assignment's place in its expert's queue, counted from one. Slot EC lies past the end of the buffers and takes every dropped assignment. `run_experts` stacks one of the three matrices of every `d2l.FeedForward` into an array of shape (E, d, w) or (E, w, d), for experts of width w, and applies the SwiGLU of Section 11.1 as three batched products, one matrix per buffer. It reads the experts' `W_g`, `W_1`, and `W_2` by name, so it assumes their default SwiGLU without biases.

The buffer has one row more than the EC slots. Every dropped assignment is written into that last row, which the experts never see, and reading the outputs back through `slot` returns the appended row of zeros for it. No group size leaves the device, so the layer never waits for the host.

Two helpers find the expert layers of a model and count the parameters that one token uses, which are all parameters except the unselected experts:

```
def moe_layers(model):
    """The MoELayer modules of a model."""
    return [m for m in model.modules() if isinstance(m, MoELayer)]

def active_params(model):
    """Parameters used per token: all but the unselected experts."""
    return d2l.num_params(model) - sum(
        d2l.num_params(l.experts) * (l.num_experts - l.num_active)
        // l.num_experts for l in moe_layers(model))
```

Since the layer maps (n, d) to (n, d) like any feed-forward network, it fits into `d2l.TransformerBlock` through the block's `ffn_factory` argument, which builds the feed-forward module of the block. We check shapes on 400 random tokens, count parameters for eight experts with two active, and look at how the router of the fresh layer spreads the 800 assignments over buffers of $\lceil 1.25 \cdot 800/8 \rceil = 125$ slots:

```
moe = MoELayer(256, num_experts=8, num_active=2)
X = torch.randn(4, 100, 256)
d2l.check_shape(moe(X), X.shape)
blk = d2l.TransformerBlock(256, num_heads=8,
                           ffn_factory=lambda: MoELayer(256, 8, 2))
d2l.check_shape(blk(X), X.shape)
print(f'{d2l.num_params(moe)} parameters stored, '
      f'{active_params(moe)} active per token')
print('shares of the assignments:',
      ' '.join(f'{s:.2f}' for s in (moe.load / moe.load.sum()).tolist()))
print(f'{float(moe.dropped) / float(moe.load.sum()):.0%} of them dropped')
```

```
4198400 parameters stored, 1051136 active per token
shares of the assignments: 0.12 0.11 0.13 0.12 0.13 0.13 0.14 0.12
0% of them dropped
```

Each expert holds $3 \cdot 256 \cdot 683 = 524{,}544$ parameters, so the layer stores about 4.2 million and a token uses about 1.05 million, a quarter. A fresh router spreads random tokens roughly evenly: no expert's share is far from the even share of $1/8$, and the buffers, a quarter larger than an even share, drop almost no assignments. Random vectors are not text, however, and the router has yet to be trained. The next part follows the shares inside a language model.

11.6.3 Routing Collapse and Load Balancing

The router and the experts learn together, and the way they learn tends to concentrate the tokens on a few experts. This part explains why and compares two remedies.

11.6.3.1 Positive Feedback in Routing

Follow the feedback loop. At initialization some expert is, by chance, slightly better than its peers on the tokens it receives. The loss gradient then rewards it twice: its parameters improve on those tokens, *and* the router's gradient raises its probability, because weighting a helpful expert more heavily lowers the loss. More tokens bring more gradient signal, which makes a better expert, which attracts more tokens. The experts that lose the early rounds receive ever fewer tokens, learn ever more slowly, and lose probability. This loop can end in *routing collapse*: a few experts receive most of the assignments, and much of the stored capacity goes unused. Shazeer et al. (2017) observed this imbalance and added balancing terms to the training loss. We describe two remedies in use today.

11.6.3.2 An Auxiliary Balancing Loss

The classic remedy adds a differentiable penalty for imbalance to the training loss. For a minibatch, let f_i be the fraction of the assignments that expert i received. It is a count, so no gradient flows through it. Let $\bar{p}_i$ be the mean of the router's probability $p_i(\mathbf{x})$ over

the tokens of the minibatch. The loss of the Switch transformer (Fedus et al. 2022), which descends from GShard's (Lepikhin et al. 2021), multiplies the two:

$$\mathcal{L}_{\text{balance}} = E \sum_{i=1}^{E} f_i \, \bar{p}_i. \tag{11.17}$$

The loss is large when the experts that receive many tokens also receive high probability. For uniform routing, $f_i = \bar{p}_i = 1/E$ and the loss equals 1. If one expert receives every token with probability close to one, it approaches E.

The balancing loss is added to the language-modeling loss with a small weight α, 0.01 in the Switch transformer. Its gradient, which flows through $\bar{p}$ only, lowers the probability of overloaded experts for every token, whatever the token needs. This is its drawback: the balancing gradient is added to the language-modeling gradient at every step, and the two conflict when some tokens do need a busy expert. With α too large, routing serves balance rather than prediction; with α too small, the imbalance persists (L. Wang et al. 2024).

11.6.3.3 Balancing Without a Loss

The conflict disappears if balance acts on the selection of experts alone and adds nothing to the loss. L. Wang et al. (2024) balance the load this way, and DeepSeek-V3 trains with their method (DeepSeek-AI, Liu, et al. 2024). A bias b_i per expert is added to the probabilities for the selection, and only for the selection. After each optimizer step, a controller moves each bias by a fixed amount u toward balance:

$$\mathcal{E}_k(\mathbf{x}) = \text{argtop}_k \left(p_i(\mathbf{x}) + b_i\right), \qquad b_i \leftarrow b_i + u \, \text{sign}(\bar{f} - f_i), \tag{11.18}$$

where $\bar{f} = 1/E$ is the even share. The bias of an overloaded expert falls, and that of an underloaded expert rises, which makes the underloaded expert slightly easier to select. Since b_i never enters the weight p_i in Equation 11.16, the model's loss contains no balancing term, and no gradient competes with the language-modeling one. The controller steers where tokens go; the gradients serve prediction alone. The update rate sets how quickly the controller reacts. DeepSeek-V3 used $u = 0.001$ for the first 14.3 trillion of its training tokens; our runs take 800 steps, and we use $u = 0.01$ so that the controller can follow the router within a few dozen steps. DeepSeek-V3 also scores experts with a sigmoid per expert instead of a softmax. Our layer keeps the softmax and changes only the selection bias, the mechanism that L. Wang et al. (2024) isolate.

11.6.3.4 Balancing Methods Compared

Both remedies fit into an ordinary training loop. The loop below is `d2l.train_steps` of Section 9.5, which draws minibatches, takes optimizer steps, and returns the training losses, with the two remedies added: `aux_weight` adds α times the sum of $\mathcal{L}_{\text{balance}}$ over the expert layers to the loss, and `bias_rate` is the rate u of the controller of Equation 11.18. The controller is a method of the layer, `balance`, which runs after each optimizer step on the load of that step's minibatch.

```
@d2l.add_to_class(MoELayer)
@torch.no_grad()  # Not a gradient step
def balance(self, rate):
    """The controller: move each bias one step toward the even load, then
    count the next batch's assignments afresh."""
    self.expert_bias += rate * torch.sign(self.load.mean() - self.load)
    self.load.zero_()

def train_moe_lm(model, data, optimizer, num_steps, aux_weight=0.0,
                 bias_rate=0.0):
    """d2l.train_steps with a balancing loss and the bias controller."""
    device = d2l.try_gpu()
    model.to(device)
    layers, losses = moe_layers(model), []
    for l in layers:
        l.reset_stats()
    while len(losses) < num_steps:
        for X, Y in data.train_dataloader():
            ce = F.cross_entropy(model(X.to(device)).flatten(0, 1),
                                 Y.to(device).flatten())
            optimizer.zero_grad()
            (ce + aux_weight * sum(l.aux_loss for l in layers)).backward()
            optimizer.step()
            for l in layers:
                l.balance(bias_rate)
            losses.append(ce.detach())
            if len(losses) == num_steps:
                return torch.stack(losses).tolist()
```

To read a trained model, we compute its validation loss with `d2l.lm_perplexity` of Section 9.5, whose logarithm is the mean cross-entropy in nats per byte, and count during that pass how many validation tokens each expert receives and how many assignments the buffers drop. The shares of each layer are returned busiest first:

```
def evaluate_moe(model, data):
    """Validation loss; each expert layer's shares of the validation
    tokens, busiest expert first; and the fraction of assignments dropped."""
    layers = moe_layers(model)
    for l in layers:
        l.reset_stats()
    loss = math.log(d2l.lm_perplexity(model, data.val_dataloader()))
    shares = [sorted((l.load / l.load.sum()).tolist(), reverse=True)
              for l in layers]
    dropped = sum(float(l.dropped) for l in layers) / sum(
        float(l.load.sum()) for l in layers)
    return loss, shares, dropped

def report(name, model, data):
    """Print the validation loss, the busiest expert's load in units of the
    even share, and the drops; return the shares."""
    val, shares, dropped = evaluate_moe(model, data)
    peak = max(l.num_experts * s[0]
```

```
                for l, s in zip(moe_layers(model), shares))
    print(f'{name:>15}: validation loss {val:.2f}, busiest expert at '
          f'{peak:.1f} times its share, {dropped:.0%} dropped')
    return shares
```

The experiments use the data and the training recipe of Section 11.2. The data are the bytes of the Wikipedia articles of `d2l.WikiText103`, about 270 million for training, in windows of 128 bytes at random offsets, and the official validation split of about 1.1 million bytes. A run almost never reads the same passage twice, so the validation loss measures what a model has learned about English rather than how much of the text it has memorized. The recipe, bundled as `d2l.LMOptimizer`, is AdamW with weight decay on the weight matrices, a learning rate that rises linearly over the first 100 steps and then follows a cosine to zero, and clipping of the gradient norm at 1. Our model is a small GPT, two blocks of width 128 with eight experts per block and one active per token, with the capacity factor of 1.25. We first read a fresh model, then train three models for 800 steps each. The runs differ only in balancing: none, the auxiliary loss at $\alpha = 0.01$, or the bias controller at $u = 0.01$. Each model draws its own initialization.

```
data = d2l.WikiText103(batch_size=64, num_steps=128)

def small_gpt(vocab_size, capacity_factor=None):
    """Two blocks of width 128; with a capacity factor, each feed-forward
    network becomes eight experts with one active per token."""
    experts = lambda: MoELayer(128, 8, 1, capacity_factor=capacity_factor)
    model = d2l.GPT(vocab_size, num_hiddens=128, num_heads=4, num_blks=2,
                    ffn_factory=None if capacity_factor is None else experts)
    return model.to(d2l.try_gpu())
```

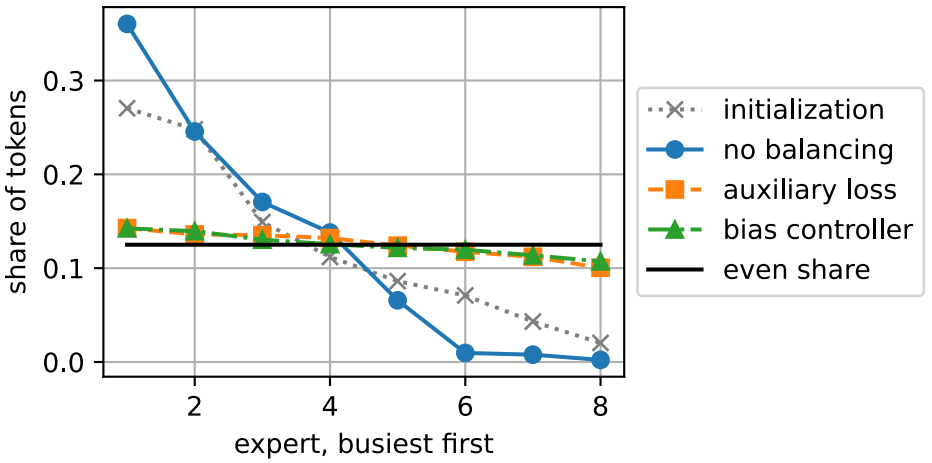
`small_gpt` builds this model, and without a capacity factor the dense GPT of the same size, which Section 11.6.4.1 uses. Each line of the output reads one model: the fresh one, then the three trained ones.

```
shares = {'initialization': report(
    'initialization', small_gpt(len(data.vocab), capacity_factor=1.25), data)}
for name, aux_weight, bias_rate in (('no balancing', 0.0, 0.0),
                                    ('auxiliary loss', 0.01, 0.0),
                                    ('bias controller', 0.0, 0.01)):
    model = small_gpt(len(data.vocab), capacity_factor=1.25)
    optimizer = d2l.LMOptimizer(model, lr=2e-3, num_steps=800, warmup=100)
    train_moe_lm(model, data, optimizer, 800, aux_weight, bias_rate)
    shares[name] = report(name, model, data)
```

```
 initialization: validation loss 5.56, busiest expert at 2.3 times its share, 22% dropped
   no balancing: validation loss 1.80, busiest expert at 3.2 times its share, 32% dropped
 auxiliary loss: validation loss 1.56, busiest expert at 1.2 times its share, 0% dropped
bias controller: validation loss 1.56, busiest expert at 1.2 times its share, 0% dropped
```

The plot shows the shares of the eight experts, sorted from the busiest to the idlest and averaged over the two layers, against the even share of $1/8$:

```
ranks = list(range(1, 9))
mean_shares = [[sum(rank) / len(rank) for rank in zip(*s)]
               for s in shares.values()]  # Mean over the layers
d2l.plot(ranks, mean_shares + [[1 / 8] * 8], 'expert, busiest first',
         'share of tokens', legend=list(shares) + ['even share'],
         fmts=('C7x:', 'C0o-', 'C1s--', 'C2^-.', 'k-'), legend_loc='outside')
```



The fresh model is unbalanced before it has seen any data, unlike the layer that we fed random vectors above, and its buffers drop many assignments. At initialization the residual stream of the first block carries little more than the embedding of the current byte. Most occurrences of a byte therefore look alike to the router and go to the same expert, and bytes are far from equally frequent: nearly one byte in five is a space. In the second block, attention has added to every position nearly the same average of the earlier bytes, and this common offset tilts most tokens toward the same expert. Training without balancing does not undo the imbalance. The routing stays concentrated: the busiest expert of a layer receives about twice its even share or more, some experts receive almost none, and the buffers keep dropping the tokens that the busy experts have no room for.

Both remedies spread the tokens much more evenly and drop few assignments. Both runs end at or below the validation loss of the unbalanced one, and the further that run has collapsed, the wider the gap. In the unbalanced model, experts that receive almost no tokens add parameters that the model hardly uses, and many tokens skip their expert altogether. At this scale neither remedy ends consistently ahead of the other. Their difference shows in larger models: L. Wang et al. (2024) trained models of 1 and 3 billion parameters both ways, and the controller balanced the load much more evenly than the auxiliary loss while reaching a slightly lower validation perplexity.

11.6.4 Mixture-of-Experts Language Models

Balancing makes the experts usable. Whether they make a better language model, and how released models arrange them, is the subject of this last part.

11.6.4.1 A Mixture-of-Experts GPT

The expert layer was built to answer one question: at the same computation per token, does a model with eight times the feed-forward parameters predict text better? Extra parameters help only a model that has more text than it can absorb, so we compare in that regime: the small GPT of the balancing experiments, two blocks of width 128, trained on about 50 million bytes, more than a hundred per parameter. The dense `d2l.GPT` competes with the same model with each feed-forward network replaced by eight experts, one active per token. The expert layers are balanced by the auxiliary loss at $\alpha = 0.01$, as in the Switch transformer, whose experiments we compare with below. Each expert is a full-width `d2l.FeedForward`, so a token passes through as many parameters in the mixture as in the dense model, apart from the router's 128×8 weights per block.

The buffers are larger than in the balancing experiments: the capacity factor is 2 rather than 1.25, another setting with which the Switch transformer was trained (Fedus et al. 2022). Bytes need the room. At $c = 1.25$ an expert's buffer holds $1.25/8 \approx 16\%$ of the assignments, yet nearly one byte in five is a space, and the first block, which sees little more than the current byte, tends to send the occurrences of a byte to the same expert. When one expert collects most of the spaces together with another frequent byte, its buffer overflows. In our runs at $c = 1.25$, the first block of some models dropped a tenth of its tokens for a thousand steps or more. The longer the overflow lasted, the smaller the mixture's final lead over the dense model, which ranged from almost nothing to five hundredths of a nat. At $c = 2$ a buffer holds a quarter of the assignments, and after the first few hundred steps almost no assignment is dropped.

```
models = {'dense': small_gpt(len(data.vocab)),
          'MoE': small_gpt(len(data.vocab), capacity_factor=2)}
for name, model in models.items():
    print(f'{name:>5}: {d2l.num_params(model) / 1e6:5.2f}M parameters, '
          f'{active_params(model) / 1e6:.2f}M active per token')
```

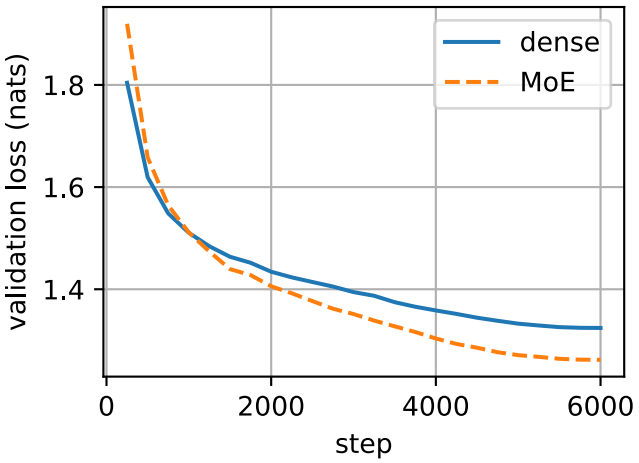
```
dense:  0.43M parameters, 0.43M active per token
  MoE:  2.26M parameters, 0.43M active per token
```

The mixture stores 5.3 times the parameters of the dense model and uses the same number per token. Both models train with the recipe of Section 11.2 for 6,000 steps, and we measure the validation loss every 250 steps:

```
num_steps, every = 6000, 250
curves = {}
for name, model in models.items():
    optimizer = d2l.LMOptimizer(model, lr=2e-3, num_steps=num_steps,
                                warmup=100)
    curves[name] = []
    for _ in range(num_steps // every):
        train_moe_lm(model, data, optimizer, every, aux_weight=0.01)
        curves[name].append(
            math.log(d2l.lm_perplexity(model, data.val_dataloader())))
    print(f'{name:>5}: validation loss {curves[name][-1]:.2f} nats')
d2l.plot(list(range(every, num_steps + 1, every)), list(curves.values()),
         'step', 'validation loss (nats)', legend=list(curves),
```

```
            fmts=('-', '--'))
```

```
dense: validation loss 1.32 nats
  MoE: validation loss 1.26 nats
```



Early in training the mixture trails the dense model: each fresh expert's output is scaled by a probability near $1/8$, so the expert layers start out much weaker than the dense feed-forward networks, and each expert learns from only an eighth of the tokens. As the dense model runs out of capacity, its curve flattens, while the mixture keeps improving. Its validation loss falls below the dense model's within the first two thousand steps and stays below it to the end, by a few hundredths of a nat. At the same computation per token, the stored experts make a better model. Large studies report the same effect. The Switch transformer reached the quality of a dense T5 model several times faster at the same number of operations per token, and it already gained from two experts (Fedus et al. 2022). Clark et al. (2022) found that routing improved models of every size they trained, from 15 million to 1.3 billion parameters, with the largest gains for the smallest models.

The gain needs data. Each expert sees about $1/E$ of the tokens, and the extra parameters help only if the text holds more than the dense model can learn from it. A corpus read many times over offers less, and ours is read less than once. Exercise 7 repeats the comparison with the six-block GPT of Section 11.2, which at 1,000 steps has more parameters than its text can fill. Krajewski et al. (2024) found that when the training tokens grow with the model, the advantage of mixtures over dense models grows with the training budget.

The gain also has costs that the operation count leaves out. The mixture keeps 5.3 times the parameters in memory, together with AdamW's two state values per parameter. Its step takes longer than its operation count suggests: the buffers hold twice as many rows as there are tokens, copying tokens into the buffers and outputs back moves data without computing, and the experts' weights are stacked anew in every step. And wherever the load is uneven, the buffers drop assignments, which a dense model never does.

11.6.4.2 Fine-Grained and Shared Experts

Released mixtures differ in how they divide a budget of expert parameters. Mixtral's experts are full copies of the feed-forward network, eight per layer and two per token. DeepSeek-

MoE (Dai et al. 2024) spends such a budget differently, in the two steps shown in Figure 11.10. Their example starts from 16 full-width experts with 2 active per token. The first step cuts each expert into m narrower ones and lets each token select m times as many: with $m = 4$, 8 of 64 experts of a quarter width instead of 2 of 16 full-width ones. Neither the stored nor the active parameter count changes. The number of combinations a token can select does, from $\binom{16}{2} = 120$ to $\binom{64}{8} \approx 4.4 \times 10^9$. A token can then assemble its network from many narrow specialists, say one for chemical vocabulary and another for nested clauses, instead of selecting two broad experts that must each cover a wide range of inputs.

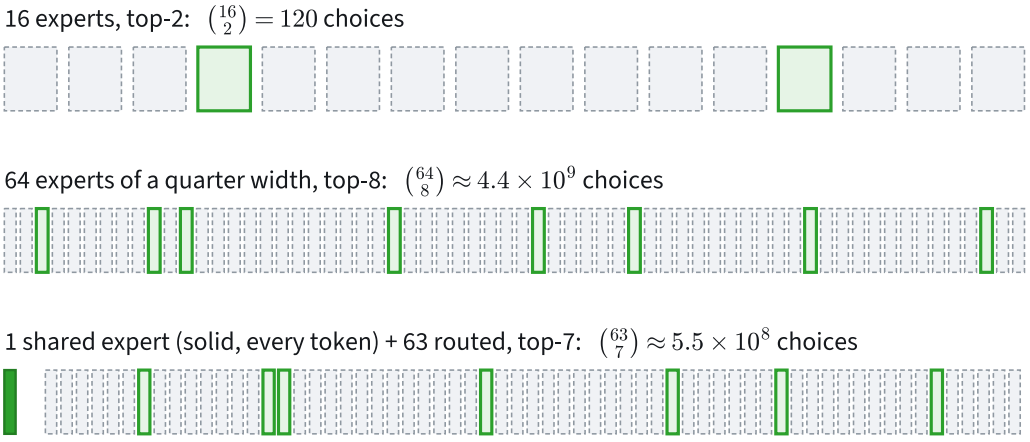


Figure 11.10: Three ways to spend the same expert parameters. Each box is an expert, its width proportional to its size, and the highlighted experts are the ones selected for one token. Top: 16 experts, of which a token uses 2. Middle: the same parameters as 64 experts of a quarter width, of which a token uses 8. Bottom: one of the narrow experts becomes a shared expert that every token uses, and each token selects 7 of the other 63. All three rows store and use the same number of parameters per token.

The second step makes one of the narrow experts *shared*: every token passes through it, in addition to its routed experts. Some processing is needed by every token, whatever its topic. With routed experts only, each expert must learn that processing for itself, so the same function is stored many times over. A shared expert stores it once and leaves the routed experts free to differ. In experiments at 2 billion stored parameters, a model with both refinements reached a lower loss on the Pile than a GShard model with the same stored and active parameters (1.81 against 1.87), and about matched a GShard model with 1.5 times the expert parameters and 1.5 times the computation (Dai et al. 2024). Neither refinement changes the logic of our layer: narrower experts need only a configurable width, and a shared expert is one more feed-forward network whose output is added to the routed mixture. The exercises build both.

Released models make different choices along these axes, as Table 11.1 shows.

Table 11.1: Routed-expert configurations of released MoE language models.

Model	Experts per layer	Active k	Expert width / d	Shared expert
Mixtral 8x7B (Jiang et al. 2024)	8	2	3.5	none
DeepSeek-V3 (DeepSeek-AI, Liu, et al. 2024)	256	8	0.29	1
Qwen3-235B (A. Yang et al. 2025)	128	8	0.38	none
gpt-oss-120b (OpenAI 2025)	128	4	1.0	none

The three later models use many more experts than Mixtral and between 4 and 8 active per token, and most of them make each expert narrower than the model itself. They disagree on the shared expert: DeepSeek's models have used shared experts since DeepSeekMoE, two in DeepSeek-V2 (DeepSeek-AI 2024) and one in DeepSeek-V3, whereas Qwen3 uses none. How many experts a model should have, how wide, and how many tokens it needs before its stored parameters help are questions of scaling, of the kind that Section 11.7 asks for dense models.

11.6.5 Summary

A dense transformer spends computation on every parameter for every token. A mixture of experts replaces each block's feed-forward network by E experts and a linear router that sends each token to k of them, weighted by the router's probabilities: the stored parameters grow with E, the computation per token with k. An efficient layer copies each token into a fixed-size buffer of each of its experts and runs all experts at once on their buffers, at the cost of dropping the assignments that do not fit. Left alone, routing concentrates on a few experts, both because similar tokens go to the same expert and because routing reinforces itself. An auxiliary loss on the product of load and probability, or a controller that biases only the selection, evens out the load. With balanced experts, a GPT with eight experts per block and one active per token reaches a lower validation loss than the dense model at the same computation per token, once the dense model has more text than it can absorb, while storing 5.3 times the parameters; released models refine the design with many narrow experts and a shared expert.

11.6.6 Exercises

1. **Router gradient.** The top-k selection in Equation 11.16 has no gradient, yet the router trains. Consider one token with router logits $\mathbf{z} = \mathbf{W}_r\mathbf{x}$, so that $\mathbf{p} = \mathrm{softmax}(\mathbf{z})$, and treat the selected set $\mathcal{E}_k$ as a constant. Write the layer output as $\mathbf{y} = \sum_{i\in\mathcal{E}_k} p_i\,\mathrm{FFN}_i(\mathbf{x})$ and let $s_i = (\partial\mathcal{L}/\partial\mathbf{y})^\top \mathrm{FFN}_i(\mathbf{x})$ be the sensitivity of the loss to expert i's output.
   1. Derive $\partial\mathcal{L}/\partial z_j$ for every expert j in terms of $\mathbf{p}$ and the s_i. Which logits receive a nonzero gradient, and on which experts' outputs does the gradient of an unselected logit depend?

2. Mixtral weights the selected experts by renormalized probabilities (Jiang et al. 2024),

$$\tilde{p}_i = \frac{p_i}{\sum_{j \in \mathcal{E}_k} p_j}, \qquad i \in \mathcal{E}_k.$$

Show that for $k = 1$ the router receives no gradient from the language-modeling loss.

3. Show that for $k = 2$ with renormalization the output depends on the logits only through the difference $z_a - z_b$ of the two selected experts a and b. Which routing decision can the router learn from the loss in this case, and which can it not?

2. **Serving arithmetic.** Qwen3-235B-A22B (A. Yang et al. 2025) has $d = 4096$ and 94 layers. Every layer holds 128 SwiGLU experts of width 1536, of which each token activates 8, and no shared expert. Its attention has 64 query heads and 4 key–value heads, each of dimension 128, and its untied embedding and output matrices cover a vocabulary of 151,936 tokens.
   1. Use `moe_accounting` for the experts and the router, and add the attention and embedding parameters as Section 11.6.1 does for Mixtral. Reproduce the 235B stored and 22B active parameters of the model's name, ignoring normalization weights.
   2. Decoding a batch of B sequences sends one token per sequence through every layer, and its speed is set by the bytes of weights read per step (Section 11.3). Assume that each token selects a uniformly random set of 8 experts, independently of the other tokens. Derive the expected number of distinct experts that a layer reads, and evaluate it for $B = 1$, 8, and 64. Above which batch size does a step read 90% of the expert weights, and which dense model then reads the same number of bytes per step?
3. **Balancing-loss gradient.** The gradient of Equation 11.17 flows through $\bar{p}_i$ only, because f_i is a count. Consider a batch of N tokens, and write $\mathbf{z}(\mathbf{x})$ for the router logits of token $\mathbf{x}$.
   1. Derive $\partial \mathcal{L}_{\text{balance}} / \partial z_j(\mathbf{x})$. For which experts j does the gradient step lower the logit of token $\mathbf{x}$, and for which does it raise it?
   2. Show that the gradient vanishes for every token whenever the batch load is uniform, $f_i = 1/E$, whatever the individual distributions $\mathbf{p}(\mathbf{x})$ are. Which property of the router does the loss therefore leave unconstrained?
   3. Show that the gradient for a token tends to zero as its router distribution approaches a one-hot vector. What does this imply for the ability of the loss to repair a collapse in which the router has become confident?
4. code **Feedback-loop ablations.** The argument of Section 11.6.3.1 names two reinforcing effects: a frequently selected expert improves on its tokens, and the router gradient raises its selection probability.
   1. Stop the gradient at the router logits inside `MoELayer`, so that $\mathbf{W}_r$ keeps its initial values while every other parameter trains. Predict whether the unbalanced run of Section 11.6.3.4 still concentrates its routing, then rerun it and compare the busiest expert's share. What still changes the routing when $\mathbf{W}_r$ is fixed?
   2. Restore the router gradient and rerun the three runs with `num_active=2`. Does the unbalanced run still concentrate, and as strongly? Use the router gradient of problem 1 to explain how a second active expert changes the signal that the runner-up receives.
5. code **Bias controller in isolation.** The controller of Equation 11.18 can be studied without training. Build `MoELayer(256, num_experts=8, num_active=1)`, draw a fixed batch

of shape $(4, 256, 256)$, that is, 1024 standard-normal tokens in four sequences, and add one fixed standard-normal vector $\mu \in \mathbb{R}^{256}$ to every token, which skews the untrained router toward a few experts. Then call `reset_stats` and repeat 500 times: run the layer on the batch, and call its `balance` method, which updates the biases and resets `load`.

1. For $u \in \{0.001, 0.01, 0.1\}$, plot the largest load $\max_i f_i$ against the iteration. How does the number of iterations needed to bring it below $1.5/E$ scale with u, and how does the load behave afterwards?
2. After the last iteration, measure the fraction of tokens whose selected expert differs from the expert with the largest $p_i(\mathbf{x})$. What determines this fraction, and how does it depend on u?
3. Replace the sign in Equation 11.18 by the raw difference, $b_i \leftarrow b_i + u'(\bar{f} - f_i)$, with u' chosen so that the largest first step equals that of $u = 0.01$. Which behavior from the first sub-problem changes?

6. **Capacity factor.** ▯ Our layer gives each expert a buffer of $C = c\,Nk/E$ slots for a batch of N tokens, as GShard and the Switch transformer do (Lepikhin et al. 2021; Fedus et al. 2022), and drops the assignments that do not fit.
   1. Without compilation, a layer could size its buffers by each batch's routing. Explain why the buffer size must still be fixed in advance when the experts live on different devices.
   2. Take $k = 1$, $N = 4096$, and a router that sends each token to a uniformly random expert, independently of the other tokens. The load of one expert is then binomial with mean N/E. Use a normal approximation, together with $\mathbb{E}[\max(Z, 0)] = 1/\sqrt{2\pi}$ for a standard normal Z, to estimate the fraction of tokens dropped at $c = 1$ for $E = 8$ and for $E = 64$. How does the fraction scale with E and N, and what does $c = 1.25$ change?
   3. Our layer fills the buffers in the order of the tokens. In a causal language model, can whether a token is dropped depend on later tokens of its own sequence? On other sequences of the minibatch? What does this imply for a model that is evaluated with batches of a different size or composition than in training, for example while decoding one token per sequence?
7. code **When the mixture loses.** Section 11.6.4.1 compares the two models where the dense model has more text than it can absorb.
   1. Repeat the comparison with the six-block GPT of width 256 from Section 11.2 and its budget of 1,000 steps, with eight full-width experts per block. Train each model three times. Which one ends lower?
   2. Compute the training bytes per dense parameter in both comparisons, and explain the outcome with the scaling laws of Section 11.7 and the data each expert sees.
   3. Train the six-block pair for longer. Does the mixture overtake the dense model, and if so, after how many steps?
8. code **Shared expert.** ▯ Write `SharedMoELayer`, a subclass of `MoELayer` that adds the output of one extra `d2l.FeedForward`, applied to every token with weight one, to the routed mixture (Section 11.6.4.2). Compare two models in the setup of Section 11.6.3.4, both balanced by the bias controller: `MoELayer` with $E = 8$ and $k = 2$, and `SharedMoELayer` with 7 routed experts and $k = 1$. Both store eight experts per layer and activate two per token.
   1. Train each model three times and compare their validation losses. Is the difference larger than the spread between the runs of one model?
   2. Dai et al. (2024) test whether a shared expert holds processing that the routed ex-

perts lack: they disable it and activate one more routed expert, which keeps the per-token computation fixed. Repeat this test on a trained `SharedMoELayer` model. Remove the shared path, set `num_active` to 2 in every layer, and compare the validation loss with that of the intact model. What does the result say about the division of labor? Why does removing the shared path without the replacement say little about it?

9. [extended] **Expert granularity.** Section 11.6.4.2 cites large-scale evidence that narrower, more numerous experts help at a fixed budget, and this section does not test it. In the setup of Section 11.6.3.4, an expert at $d = 128$ has SwiGLU width 341. Give `MoELayer` experts of configurable width, and compare $(E, k) \in \{(8, 1), (16, 2), (32, 4), (64, 8)\}$ with expert widths 341, 171, 85, and 43, all balanced by the bias controller. These configurations store and activate the same expert parameters as $(8, 1)$ to within 1%, while the number of expert combinations available to a token grows from 8 to about 4.4×10^9.
   1. Train each configuration three times and compare the validation losses. Does finer granularity help at this scale by more than the spread between runs? Relate the outcome to the ablations of Dai et al. (2024) .
   2. The argument of Section 11.6.4.2 assumes that tokens use many different combinations. In one layer of each trained model, record the selected expert set of every token in a fixed set of validation batches, and count the distinct sets that occur. Compare the count with the number of tokens and with the number of available combinations. How much of the added flexibility does the model use?

11.7 Scaling Laws

Every model so far in this chapter was sized by hand. We chose a width, a depth and a number of training steps, and if the result had disappointed us, we could have changed them and trained again. A large model offers no second try. The GPT of Section 11.2 trained on about 2×10^{14} floating-point operations. The largest Llama 3 model used 3.8×10^{25} (Grattafiori et al. 2024), about 10^{11} times as many, in a single run. How should such a budget be divided, before the run starts, between the parameters of the model and the tokens it reads?

The answer rests on two regularities that can be measured on runs a million times smaller. The cost of a run follows from the shape of the model by counting. And the loss falls smoothly with the number of parameters and the amount of data, along a power law that can be fitted on small models and extended to large ones. Such a fit is called a *scaling law*.

This section derives the cost, about $6ND$ operations for N parameters and D training tokens, and checks it against the framework's own count. It then trains fifteen small GPTs, five sizes at three data budgets, to see how the loss depends on each. Finally it takes the law fitted by Hoffmann et al. (2022) to hundreds of larger runs, derives how a fixed budget should be split between parameters and tokens, and compares that split with what deployed models do. Section 9.9 transferred the learning rate from one model size to another; here the model size itself is the variable.

```
from torch.nn.attention import SDPBackend, sdpa_kernel
from torch.utils.flop_counter import FlopCounterMode
```

11.7.1 Counting Parameters and FLOPs

11.7.1.1 Blocks and Embeddings

The cost of a run depends on how many parameters the model has and where they sit, so we begin by counting them. All models in this section are `d2l.GPT`s of Section 11.2, built through one shorthand:

```
gpt = d2l.GPT
```

A GPT holds two kinds of parameters, and they grow differently. Each block holds about $12d^2$ for width d: $4d^2$ in the attention projections $\mathbf{W}_q$, $\mathbf{W}_k$, $\mathbf{W}_v$, $\mathbf{W}_o$, and $8d^2$ in the feed-forward network, as two matrices of width $4d$ with GELU or three of width $8d/3$ with SwiGLU (Section 11.1). A stack of L blocks thus holds $12Ld^2$. The token embedding holds Vd for a vocabulary of V tokens; weight tying uses it a second time as the output head without storing a copy. A learned position table adds max_len $\times\, d$. The cell below counts both parts for our byte-level model and for GPT-2 small:

```
def census(model):
    L, d = len(model.blks), model.num_hiddens
    emb = d2l.num_params(model.token_emb) + (
        d2l.num_params(model.pos_emb) if model.pos == 'learned' else 0)
    total = d2l.num_params(model)
    print(f'embedding {emb / 1e6:6.2f}M ({emb / total:4.0%}),  '
          f'blocks {(total - emb) / 1e6:6.2f}M '
          f'= {(total - emb) / (L * d ** 2):.2f} L d^2,  '
          f'total {total / 1e6:.2f}M')

census(gpt(256))
census(gpt(50257, num_hiddens=768, num_heads=12, num_blks=12,
           pos='learned', norm='layer', act='gelu', bias=True))
```

```
embedding   0.07M (  1%),  blocks   4.72M = 12.01 L d^2,  total 4.79M
embedding  39.38M ( 32%),  blocks  85.06M = 12.02 L d^2,  total 124.44M
```

Both rows confirm the $12Ld^2$ rule to a fraction of a percent; the residue is normalization gains and, for GPT-2, biases. The two models differ in the share of the embedding. It is 1.4% of our model with its 256 bytes, but nearly a third of GPT-2, whose 124 million parameters are an 85-million-parameter transformer attached to lookup tables of 39 million entries, 38.6 million for the tokens and 0.8 million for the positions. Comparing Vd with $12Ld^2$ shows that the embedding dominates when $V \gtrsim 12Ld$, that is, for small models with large vocabularies. The size of the table follows the tokenizer, not the computation each token receives, so Kaplan et al. (2020) counted only *non-embedding* parameters. The miniature study below does the same, so there N means the number of non-embedding parameters. The law of Hoffmann et al. (2022) that follows it counts all parameters.

11.7.1.2 Six FLOPs per Parameter and Token

Multiplying a vector by a matrix $\mathbf{W} \in \mathbb{R}^{m \times n}$ takes mn multiplications and mn additions, 2 floating-point operations (FLOPs) per weight. In the forward pass every weight of a linear layer meets every token once, so the forward pass costs about $2N$ FLOPs per token. The backward pass costs twice as much. A linear layer $\mathbf{y} = \mathbf{W}\mathbf{x}$ receives the gradient δ with

respect to its output and computes two products from it: the gradient $\mathbf{W}^\top \delta$ with respect to its input, which continues the chain rule into the layer below, and the gradient $\delta \mathbf{x}^\top$ with respect to its weights, which the optimizer uses. Each costs as much as the forward product. Training on D tokens therefore takes

$$C \approx \underbrace{2ND}_{\text{forward}} + \underbrace{4ND}_{\text{backward}} = 6ND \tag{11.19}$$

FLOPs (Kaplan et al. 2020), the estimate that Section 11.2 used to place its run. Generating a token without training needs only the forward pass, about $2N$ FLOPs (Section 11.3).

The count leaves out all work that is not a product with a weight matrix. The largest such term is attention itself. For a context of n tokens, each query takes a dot product with n keys and averages n values, about $4nd$ FLOPs per token and block in the forward pass, against $24d^2$ for the block's $12d^2$ weights. The ratio of the omitted to the counted work is $n/(6d)$: 8% for our models, with $n = 128$ and $d = 256$, and 100% for a model of width 4,096 that reads 24,576 tokens at once. Normalization, softmax, activation functions, and the optimizer update add a few operations per element, far fewer than the matrix products at these widths. For dense transformers whose context is short compared with $6d$, $6ND$ is the leading term of the cost.

11.7.1.3 Checking the Arithmetic

We compare the estimate with the framework's own count of the operations in one training step on a minibatch of 64 windows of 128 tokens. The model is the default GPT with a GELU feed-forward network, which holds the same $12d^2$ per block as SwiGLU in matrices of round sizes. The cell prints the counted forward cost against $2ND$, the counted cost of the whole step against $6ND$, and the ratio of backward to forward work. Here N counts all two-dimensional parameters, so that it includes the tied embedding in its role as the output head.

`FlopCounterMode` intercepts every operator call and adds up the FLOPs of matrix products; it counts nothing for elementwise operations. The fused attention kernel that PyTorch picks on a CPU has no counting formula, so the cell selects the math backend with `sdpa_kernel`. That backend writes attention as two batched matrix products, which the counter sees, and it counts the full $n \times n$ score matrix, the masked half included.

```
model = gpt(256, act='gelu')
X = torch.randint(0, 256, (64, 128))
Y = torch.randint(0, 256, (64, 128))
N = sum(p.numel() for p in model.parameters() if p.ndim == 2)
with sdpa_kernel(SDPBackend.MATH):  # Attention as visible matrix products
    with FlopCounterMode(display=False) as fwd:
        loss = F.cross_entropy(model(X).flatten(0, 1), Y.flatten())
    with FlopCounterMode(display=False) as bwd:
        loss.backward()
f, b = fwd.get_total_flops(), bwd.get_total_flops()
print(f'forward {f:.3e} FLOPs, {f / (2 * N * X.numel()):.3f} x 2ND')
print(f'step    {f + b:.3e} FLOPs, {(f + b) / (6 * N * X.numel()):.3f} x 6ND')
print(f'backward / forward = {b / f:.2f}')
```

```
forward 8.483e+10 FLOPs, 1.082 x 2ND
step    2.545e+11 FLOPs, 1.082 x 6ND
backward / forward = 2.00
```

The counted cost lies 8 to 12% above the estimate, about the size of the $n/(6d)$ correction for the attention scores, and the backward pass costs twice the forward pass, as the derivation says. A count of FLOPs also converts into time. A run of C FLOPs on hardware that sustains R FLOPs per second takes C/R seconds, and the ratio of R to the hardware's peak is its *utilization*. Models as small as ours reach only a small fraction of the peak of a GPU, because their matrices are too small to keep it busy; Chapter 13 measures this fraction and raises it.

11.7.2 A Miniature Scaling Study

Counting gives the cost of a run but not the loss it reaches. Across many orders of magnitude, the loss of a language model falls as a power law in the number of parameters, in the amount of data, and in compute, as long as neither of the other two limits it (Kaplan et al. 2020). On log–log axes a power law $L = aN^{-\alpha}$ is a straight line with slope $-\alpha$, so a line fitted to small models predicts the loss of larger ones. We now measure this structure in miniature, and in particular the point where the line stops: when does a larger model stop helping?

11.7.2.1 Five Sizes and Three Data Budgets

To separate the effect of model size from that of data, we train a grid of models that varies the two independently. The corpus is the one of Section 11.2, the bytes of the Wikipedia articles of `d2l.WikiText103`: about 270 million training bytes, read in windows of 128 at random offsets, and the official validation split of about 1.1 million bytes.

```
data = d2l.WikiText103(batch_size=64, num_steps=128)
print(f'{len(data.train) / 1e6:.0f}M training bytes, '
      f'{len(data.val) / 1e6:.2f}M validation bytes, '
      f'vocabulary of {len(data.vocab)}')
```

```
270M training bytes, 1.14M validation bytes, vocabulary of 256
```

The grid has two axes. Five sizes of `d2l.GPT` grow width and depth together, as model families do: widths from 96 to 384 and depths from 3 to 8 blocks, or 0.33 to 14.2 million non-embedding parameters, a factor of 43. Each size trains at three data budgets: 500, 1,000, and 2,000 steps of 64 windows, that is D = 4.1, 8.2, and 16.4 million tokens. Even the largest budget reads about 6% of the training text, so every budget consists of fresh text, and the grid varies the amount of data without repeating any of it.

The training recipe is that of Section 11.2 (`d2l.LMOptimizer`): AdamW with weight decay 0.1 on the weight matrices, a linear warmup over the first tenth of the steps, a cosine decay of the learning rate to zero, and gradient clipping. Two of its settings depend on the run, and both matter for a scaling study. First, every run has its own schedule, which decays to zero at the end of that run's budget. Reading the loss after 500 steps off a run scheduled for 2,000 would measure a model whose learning rate is still high, and Hoffmann et al. (2022) found that this choice made earlier fits overestimate the loss of short runs. Second, the learning rate follows the width. Section 9.9 showed that the largest stable Adam learning rate shrinks in proportion to $1/d$, so a rate tuned at one width undertrains or destabilizes the others (exercise 5 measures the effect). We anchor the peak rate at 2×10^{-3} for width 256, the rate of Section 11.2, and scale it as $256/d$. There is no dropout: no run reads any

passage twice. Each line of the output is one size, with its validation losses at the three budgets and its training loss over the last 50 steps of the longest run:

```
sizes = ((96, 3, 4), (128, 3, 4), (192, 4, 6), (256, 6, 8), (384, 8, 8))
budgets = (500, 1000, 2000)  # Steps of 64 windows of 128 bytes
Ns, train, val = [], [], {num_steps: [] for num_steps in budgets}
for d, num_blks, num_heads in sizes:
    for num_steps in budgets:
        model = gpt(len(data.vocab), num_hiddens=d, num_heads=num_heads,
                    num_blks=num_blks)
        optimizer = d2l.LMOptimizer(model, lr=2e-3 * 256 / d,
                                    num_steps=num_steps,
                                    warmup=num_steps // 10)
        losses = d2l.train_steps(model, data, optimizer, num_steps)
        val[num_steps].append(
            math.log(d2l.lm_perplexity(model, data.val_dataloader())))
    # The size of the models just trained, and the training loss of the
    # longest of their runs
    Ns.append(d2l.num_params(model) - d2l.num_params(model.token_emb))
    train.append(sum(losses[-50:]) / 50)
    print(f'd = {d:3d}, {num_blks} blocks, N = {Ns[-1] / 1e6:5.2f}M: '
          'validation ' + ' / '.join(f'{val[s][-1]:.2f}' for s in budgets)
          + f', training {train[-1]:.2f}')
```

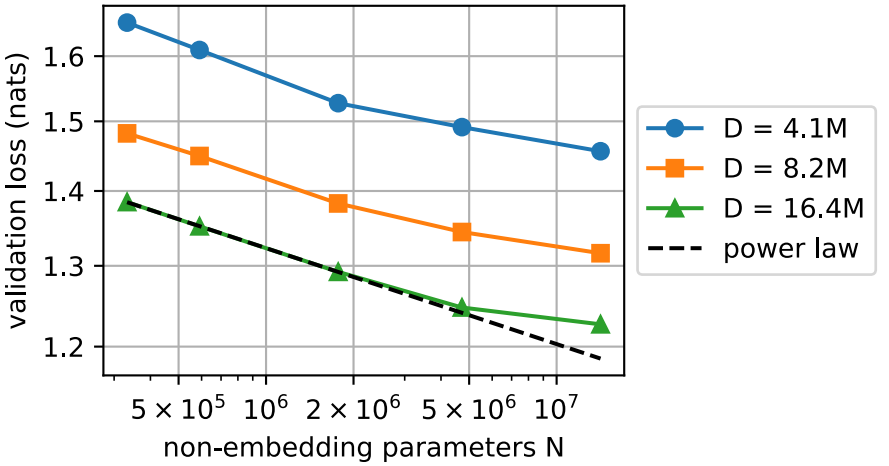
```
d =  96, 3 blocks, N =  0.33M: validation 1.65 / 1.48 / 1.38, training 1.41
d = 128, 3 blocks, N =  0.59M: validation 1.61 / 1.45 / 1.35, training 1.38
d = 192, 4 blocks, N =  1.77M: validation 1.53 / 1.38 / 1.29, training 1.32
d = 256, 6 blocks, N =  4.72M: validation 1.49 / 1.34 / 1.25, training 1.27
d = 384, 8 blocks, N = 14.16M: validation 1.46 / 1.32 / 1.23, training 1.24
```

11.7.2.2 Reading the Curves

The plot shows the validation loss against N on log–log axes, one curve per data budget. The dashed line is a power law fitted to the three smallest models at the largest budget and extended over the whole range: the prediction that a pure power law would make for the two larger models.

```
x = np.log(np.array(Ns))
slope, intercept = np.polyfit(x[:3], np.log(val[budgets[-1]][:3]), 1)
line = np.exp(intercept + slope * x)
d2l.plot(Ns, [val[s] for s in budgets] + [line],
         'non-embedding parameters N', 'validation loss (nats)',
         legend=[f'D = {s * 64 * 128 / 1e6:.1f}M' for s in budgets]
         + ['power law'], xscale='log', yscale='log',
         fmts=('C0o-', 'C1s-', 'C2^-', 'k--'), legend_loc='outside')
print(f'slope {slope:.3f}; the largest model lies '
      f'{val[budgets[-1]][-1] - line[-1]:.3f} nats above the line')
```

```
slope -0.041; the largest model lies 0.041 nats above the line
```



At every budget the validation loss falls as the model grows, and more data lowers every curve. The three smallest models at the largest budget lie close to a straight line, but the two larger models fall short of its extension: the curve flattens. At every budget, the largest model gains only a few hundredths of a nat over the next one, although it has three times the parameters. More parameters lower the loss only as long as there is data for them to learn from. With the data fixed, the loss approaches a floor that more parameters do not lower, and more data lowers the floor.

Two checks support this reading. Each point is a single run with fresh random weights, and rerunning moves most points by a hundredth or two of a nat, less than the gaps between the curves. And no run reads the same text twice, so a larger budget means more distinct text, not more passes over the same text. The training loss of the longest runs stays within a few hundredths of the validation loss, so the models do not memorize. Repetition would change these results only slowly: Muennighoff et al. (2023) trained models of up to 9 billion parameters on repeated data and found that up to about four passes give almost the same loss as fresh data, with each further pass worth less and less.

11.7.3 Compute-Optimal Allocation

11.7.3.1 The Chinchilla Law

Our miniature study shows the shape of the loss, but over too small a range to plan a large run. That needs a law fitted to runs at scale. Hoffmann et al. (2022) trained over 400 models, from 70 million to over 16 billion parameters on 5 to 500 billion tokens, and fitted their final losses with

$$L(N, D) = E + \frac{A}{N^{\alpha}} + \frac{B}{D^{\beta}}. \tag{11.20}$$

The three terms separate what limits the loss. E is the loss of an ideal model on the training distribution, the entropy of the text that no budget removes. A/N^{α} is the excess loss of a model with N parameters trained on unlimited data, and B/D^{β} the excess loss of an unlimited model trained on D tokens. Here N counts all parameters, embeddings included. The choice matters at small scale, where the embedding is a large share of a model, because

it changes the fitted exponent: Pearce and Song (2024) trace to it most of the reason why the earlier fit of Kaplan et al. (2020) favored larger models (exercise 3 measures the effect on a slope).

In our grid the two counts differ little: the table of 256 bytes adds 7% to the smallest model and under 1% to the largest, which changes the slope on log–log axes by about 2%, so the law can be compared with the grid almost directly. With D fixed, $E + B/D^{\beta}$ is a constant, and only A/N^{α} falls as the model grows. A constant plus a power law is straight on log–log axes only while the power law dominates; once A/N^{α} has shrunk to the size of the constant, the curve flattens toward $E + B/D^{\beta}$. A smaller D raises that constant, so the curve flattens earlier. The law counts D repeated tokens as D fresh ones. Our grid never tests this assumption, which holds for a few passes but not for many more.

11.7.3.2 The Optimal Split

A fixed budget $C \approx 6ND$ allows a large model trained briefly or a small model trained long. To find the best split, substitute $D = C/(6N)$ into Equation 11.20 and set the derivative with respect to N to zero. With $\partial D/\partial N = -D/N$, the optimum satisfies

$$\alpha \frac{A}{N^{\alpha}} = \beta \frac{B}{D^{\beta}}. \tag{11.21}$$

At the optimum the two terms that the budget can shrink fall at equal marginal rates, so moving FLOPs from parameters to data, or back, gains nothing. Solving Equation 11.21 together with $ND = C/6$ gives

$$N^{*} = G \left(\frac{C}{6}\right)^{\beta/(\alpha+\beta)}, \quad D^{*} = \frac{1}{G}\left(\frac{C}{6}\right)^{\alpha/(\alpha+\beta)}, \quad \text{where } G = \left(\frac{\alpha A}{\beta B}\right)^{1/(\alpha+\beta)}. \tag{11.22}$$

When α and β are about equal, both N^{*} and D^{*} grow like $C^{1/2}$: a budget a hundred times larger gives a model ten times larger, trained on ten times as many tokens, and the ratio D^{*}/N^{*} hardly changes with the budget. Figure 11.11 draws the geometry. A budget of equal compute is a straight line of slope -1 on log–log axes, and the compute-optimal model is the point where that line touches the lowest loss contour it reaches. Moving along the line in either direction crosses to higher loss.

The constants come from fits, and the fits disagree. Hoffmann et al. (2022) report $E = 1.69$, $A = 406.4$, $B = 410.7$, $\alpha = 0.34$, $\beta = 0.28$. Besiroglu et al. (2024) reconstructed the data from the paper's figures, fitted the same form again, and obtained $E = 1.82$, $A = 482.0$, $B = 2085.4$, $\alpha = 0.348$, $\beta = 0.366$. They found the original constants inconsistent with the paper's two other estimates of the optimum and their reported confidence intervals far too narrow for the number of runs. The cell evaluates Equation 11.22 for both fits at Chinchilla's own budget, 5.76×10^{23} FLOPs, at which Chinchilla trained 70 billion parameters on 1.4 trillion tokens, 20 tokens per parameter:

```
fits = {'Hoffmann et al.': (1.69, 406.4, 410.7, 0.34, 0.28),
        'Besiroglu et al.': (1.82, 482.0, 2085.4, 0.348, 0.366)}

def optimum(C, E, A, B, alpha, beta):
```

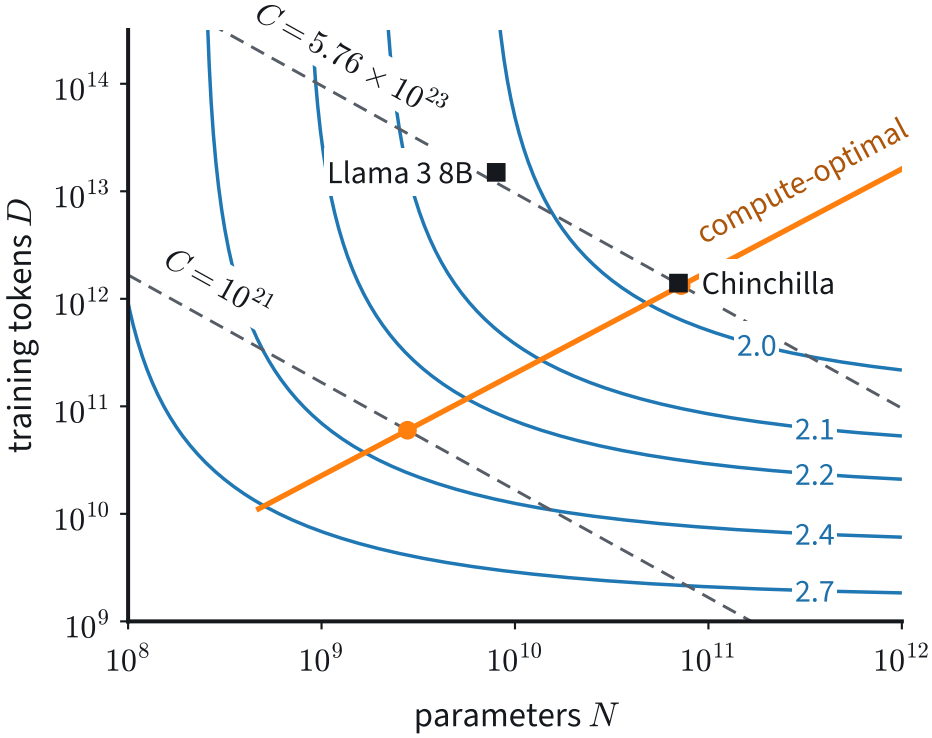


Figure 11.11: Loss contours of the Chinchilla law with the refit constants of Besiroglu et al. (2024), over parameters and training tokens. Dashed: budgets of equal compute $C = 6ND$; dots: the compute-optimal model of each budget; orange: the compute-optimal path. Chinchilla lies on the path; Llama 3 8B, on a budget about a quarter larger, lies far above it.

```
    G = (alpha * A / (beta * B)) ** (1 / (alpha + beta))
    N = G * (C / 6) ** (beta / (alpha + beta))
    return N, C / (6 * N)

for name, fit in fits.items():
    N, D = optimum(5.76e23, *fit)
    print(f'{name:17s} N* = {N / 1e9:3.0f}B, D* = {D / 1e12:.2f}T, '
          f'{D / N:3.0f} tokens per parameter')
```

```
Hoffmann et al.   N* =  32B, D* = 2.98T,  93 tokens per parameter
Besiroglu et al.  N* =  72B, D* = 1.33T,  18 tokens per parameter
```

Only the refit reproduces the allocation Chinchilla was trained with. The published constants put the optimum at less than half the model size and twice the data, about 90 tokens per parameter. Under the refit, the two exponents are nearly equal, so N^* and D^* both grow almost exactly like $C^{1/2}$ and the optimum stays near 20 tokens per parameter over many orders of magnitude of compute. This ratio belongs to that study's data, tokenizer, and model family; a different corpus or architecture moves it.

11.7.3.3 Training Past the Optimum

Compute-optimal means the lowest loss for a given training budget. A deployed model also costs about $2N$ FLOPs for every token it generates, so a smaller model trained on more tokens can be cheaper over its lifetime, although its training reaches the same loss less efficiently (Sardana et al. 2024). The cell applies the refit to dense models with published sizes and token counts, as if they had been trained on Chinchilla's data. For each it prints the tokens per parameter, the model's size relative to the compute-optimal model of the same training budget, and the extra loss that the law predicts for the difference:

```
E, A, B, alpha, beta = fits['Besiroglu et al.']
loss = lambda N, D: E + A / N ** alpha + B / D ** beta
reported = {'Chinchilla': (70e9, 1.4e12), 'Llama 3 8B': (8e9, 15e12),
            'Llama 3 405B': (405e9, 15.6e12), 'OLMo 2 7B': (7e9, 4.05e12),
            'Gemma 3 27B': (27e9, 14e12)}
for name, (N, D) in reported.items():
    N_opt, D_opt = optimum(6 * N * D, E, A, B, alpha, beta)
    print(f'{name:12s} {D / N:5.0f} tokens per parameter, '
          f'{N / N_opt:4.2f} x optimal size, '
          f'loss +{loss(N, D) - loss(N_opt, D_opt):.3f}')
```

```
Chinchilla      20 tokens per parameter, 0.96 x optimal size, loss +0.000
Llama 3 8B    1875 tokens per parameter, 0.10 x optimal size, loss +0.054
Llama 3 405B    39 tokens per parameter, 0.65 x optimal size, loss +0.001
OLMo 2 7B      579 tokens per parameter, 0.18 x optimal size, loss +0.037
Gemma 3 27B    519 tokens per parameter, 0.18 x optimal size, loss +0.023
```

The largest Llama 3 model sits close to the optimum; Meta's own fit on its data put the optimum for that budget at 402 billion parameters and 16.55 trillion tokens (Grattafiori et al. 2024). The smaller models are trained far past it, by design. Llama 3 8B reads almost 1,900 tokens per parameter. The law puts it at a tenth of the compute-optimal size and predicts about 0.05 nats of extra loss. Each token it generates, however, takes a tenth of the computation that the optimal model would need. For a model that will serve trillions of tokens, the

smaller model is the better choice. Choosing such budgets, and the further cost of running out of fresh data, are topics of the Language Models part (Chapter 43). A mixture of experts (Section 11.6) does not fit this accounting: its training cost follows its active parameters, its capacity all of them, and the law above was fitted to dense models only.

11.7.4 Configurations of Deployed Models

The models trained on such budgets are built from the components of this chapter. Table 11.2 compiles the architecture sections of seven open-weights families: Mistral 7B (Jiang et al. 2023), Llama 3 (Grattafiori et al. 2024), Qwen3 (A. Yang et al. 2025), OLMo 2 and 3 (Team OLMo et al. 2025; Team OLMo 2025), DeepSeek-V3 (DeepSeek-AI, Liu, et al. 2024), Gemma 3 (Gemma Team 2025), and GPT-OSS (OpenAI 2025).

Table 11.2: Architectures of seven open-weights model families, as reported in their technical reports (2023–2025). Attention and cache: Section 11.3. Normalization: Section 11.1. Positions: Section 10.4. Mixture of experts: Section 11.6.

model	attention and cache	normalization	positions	FFN	dense or MoE
Mistral 7B (2023)	GQA $32{:}8$; sliding window 4096	RMSNorm, pre	RoPE, $\theta = 10^4$	SwiGLU	dense
Llama 3 (2024)	GQA $32{:}8$ (8B), $64{:}8$ (70B), $128{:}8$ (405B)	RMSNorm, pre	RoPE, $\theta = 5 \times 10^5$	SwiGLU	dense
Qwen3 (2025)	GQA: $64{:}8$ dense (32B), $64{:}4$ MoE (235B-A22B)	RMSNorm, pre; QK-norm	RoPE, $\theta = 10^6$; YaRN for long context	SwiGLU	dense, and MoE: 128 experts, 8 active
OLMo 2/3 (2024/25)	MHA (7B), GQA (32B); OLMo 3: window 4096 on 3 of 4 layers	RMSNorm on sublayer outputs; QK-norm	RoPE	SwiGLU	dense
DeepSeek-V3 (2024)	MLA: KV compressed to a 512-dim latent	RMSNorm, pre	RoPE on a decoupled 64-dim slice per head	SwiGLU	MoE: 256 experts plus 1 shared, 8 active
Gemma 3 (2025)	GQA; local:global $5{:}1$, window 1024	RMSNorm, pre and post; QK-norm	RoPE, $\theta = 10^6$ global, 10^4 local	GeGLU	dense

model	attention and cache	normalization	positions	FFN	dense or MoE
GPT-OSS (2025)	GQA 64:8; window 128 on alternate layers; learned sinks	RMSNorm, pre	RoPE; YaRN	SwiGLU, clamped	MoE: 128 or 32 experts, 4 active

Every row is the block of Section 11.1 in the causal wiring of Section 11.2, and the rows agree on most of its settings: a gated feed-forward network, RoPE, RMSNorm with an unnormalized residual stream, no biases in most rows, and no dropout in any. (OLMo normalizes each sublayer's output before it joins the stream rather than its input, which also leaves the stream itself unnormalized.) The absence of dropout is a consequence of scale: a model that reads trillions of tokens about once has nothing to memorize. The rows differ on two axes. The first is the size of the KV cache: grouped-query attention in most rows, sliding windows on some layers, and a compressed latent in DeepSeek-V3, all from Section 11.3. The second is capacity per FLOP: three of the seven families use a mixture of experts (Section 11.6). The newest designs depart further. Hybrid models replace most attention layers with the linear recurrences of Chapter 12 (Kimi Team 2025b), and Llama 4 removes positional encoding from some of its attention layers (Meta AI 2025).

At width 256 and six blocks, three rows of the table become calls of our constructor. The modern rows need only the number of key–value heads, because the defaults of `d2l.GPT` are the recipe the table shares. Llama 3 8B shares each key–value head among 4 query heads, Qwen3-32B among 8. The cell prints the parameter count and the bytes of KV cache per token in 32-bit floats:

```
recipes = {'GPT-2': dict(pos='learned', norm='layer', act='gelu',
                         bias=True),
           'Llama 3 8B': dict(num_kv_heads=2),
           'Qwen3-32B': dict(num_kv_heads=1)}
for name, cfg in recipes.items():
    model = gpt(1024, num_hiddens=256, num_heads=8, num_blks=6, **cfg)
    d2l.check_shape(model(d2l.zeros((2, 16), dtype=d2l.int32)),
                    (2, 16, 1024))
    print(f'{name:10s} {d2l.num_params(model) / 1e6:.2f}M parameters, '
          f'{d2l.kv_cache_bytes(model, 1) / 1024:4.1f} KiB of cache '
          'per token')
```

```
GPT-2      5.26M parameters, 12.0 KiB of cache per token
Llama 3 8B 4.40M parameters,  3.0 KiB of cache per token
Qwen3-32B  4.30M parameters,  1.5 KiB of cache per token
```

Sharing key–value heads removes part of the key and value projections, a tenth of the parameters or so at this width, but it shrinks the cache by the full factor of the sharing, 4 and 8. The settings the flags do not reach would enter as new attention modules. QK-norm (Section 11.1.2.4) enters through the `attn_factory` hook of `TransformerBlock`, as in an exercise of Section 11.1. A sliding window is a mask (Section 10.5, Section 11.3). A learned latent cache replaces the stored keys and values, which Section 11.3 sketches with a fixed basis. The ex-

pert layers of the MoE rows enter through `ffn_factory` (Section 11.6). Deployed models are thus configurations of the chapter's one block, trained at budgets that the law of this section helps to plan.

11.7.5 Summary

Training a dense transformer with N parameters on D tokens costs about $6ND$ FLOPs, two per parameter and token in the forward pass and four in the backward pass; attention scores add a fraction $n/(6d)$ for contexts of n tokens, which the framework counts confirm. In a grid of small GPTs on a fixed corpus, the loss falls as a power law in N until it approaches a floor that only more data lowers. The law $L = E + A/N^{\alpha} + B/D^{\beta}$ explains this floor and, under a budget $C \approx 6ND$, gives the compute-optimal split: parameters and tokens both grow about like $C^{1/2}$, at about 20 tokens per parameter in the refit of the Chinchilla data. Smaller deployed models are trained far past this point to make generation cheaper, and all of them share one block whose settings differ mainly in the KV cache and in the use of experts.

11.7.6 Exercises

1. **Attention-score correction.** Equation 11.19 omits the attention-score and value-mixing work, about $4nd$ FLOPs per token and block in the forward pass for a context of n tokens, against about $24d^2$ for the block's linear maps.
    1. For the minibatch of Section 11.7.1.3 ($n = 128$, $d = 256$, six blocks, eight heads, 64×128 tokens), compute the attention term and add it to $2ND$. Compare the sum with the forward count that your framework printed. If a remainder is left, express it in FLOPs per token and block. Then estimate the elementwise work of one block (the RoPE rotation of queries and keys, the attention softmax, the GELU of the $4d$-wide hidden layer, two normalizations, and two residual additions) at a few FLOPs per element, and check whether it accounts for the remainder to within a factor of two.
    2. A fused causal kernel skips the masked half of the score matrix. Llama 3 8B has width $d = 4096$, was pretrained at context 8,192, and was later extended to 131,072 (Grattafiori et al. 2024). Treat each of its blocks as $12d^2$ parameters. By what factor does $6ND$ underestimate the training cost per token at each context length, with and without causal skipping?
2. **Variations on six.** The factor $6 = 2 + 4$ in Equation 11.19 assumes that every linear layer computes its forward product, the gradient with respect to its input, and the gradient with respect to its weights, each once. Ignore the embedding and the output head, and give the training cost per token in multiples of N for each of the following variants.
    1. Activation checkpointing (Section 5.7.3.3) stores only each block's input and recomputes the block's forward pass during the backward pass.
    2. The bottom half of the blocks is frozen, and only the top half is trained.
    3. The top half of the blocks is frozen, and only the bottom half is trained. Explain why this answer differs from the previous one.
3. **Total versus non-embedding parameters.** Suppose that each model in `sizes` had used GPT-2's vocabulary of $V = 50{,}257$ tokens with tied embeddings and had reached the validation losses it reached at the largest budget with the 256-byte vocabulary.

1. Compute the total parameter count of each model and the ratio between the largest and the smallest. Compare this ratio with the one for the non-embedding counts in `Ns`.
2. Take two models with non-embedding counts $N_1 < N_2$ and total counts $T_1 < T_2$. Show that the log–log slope of loss against total count equals the slope against non-embedding count multiplied by $\log(N_2/N_1)/\log(T_2/T_1)$, and evaluate this factor for the smallest and the third-smallest model.
3. Repeat the evaluation for Llama 3's vocabulary of 128,256 tokens. Which of the three exponents would you use to predict the loss of a much larger model, one whose embedding is a negligible share of its parameters, and why do the other two mislead?

4. **Pricing a mixture-of-experts run.** DeepSeek-V3 (DeepSeek-AI, Liu, et al. 2024) has 671B parameters, of which 37B are active for each token, and was pretrained on 14.8T tokens in 2.664M H800 GPU hours. Take the H800's dense peak as about 990 TFLOP/s in BF16 and twice that in FP8, the precision in which the run executes the matrix multiplications of its linear layers.

   1. Which of the two parameter counts belongs in Equation 11.19, and why? Estimate the pretraining FLOPs, and compare them with those of a dense 671B model trained on the same tokens.
   2. Compute the utilization implied by your estimate against each peak. Name two kinds of work in this run that the estimate omits, so that the true utilization is higher.

5. code **Frozen learning rate.** The sweep of Section 11.7.2.1 scales the peak learning rate as $2 \times 10^{-3} \cdot 256/d$.

   1. Using the width dependence of the tuned learning rate in Section 9.9, predict which points of the largest budget move, and in which direction, when every model uses 2×10^{-3}. Then rerun that column of the sweep with this rate.
   2. Rerun one model of the original sweep twice to measure the run-to-run variation. By how much does the frozen rate change the validation loss of each model, compared with this variation? Which part of the shape of the frozen-rate curve is a tuning artifact?

6. code **A sixth size.** ⊠ A sixth model for the sweep of Section 11.7.2.1 has width 512, ten blocks, and eight heads, about 31.5M non-embedding parameters.

   1. Before training it, predict its validation loss at the largest budget twice: by extending the dashed power law, and by assuming that the corpus allows no improvement over the largest measured model. Estimate the run's cost with Equation 11.19.
   2. Train it with the same recipe at all three budgets. Which prediction was closer, and what do the three results say about the headroom that a budget of 16.4M tokens leaves?

7. code **Fitting the law to the grid.** ⊠ Fit Equation 11.20 to the fifteen validation losses of the sweep, with D in tokens, by minimizing the squared error of $\log L$ over E, $\log A$, $\log B$, α, and β with a gradient-based optimizer from several starting points.

   1. Report the fitted constants and the largest residual. Is the fitted E below every measured loss?

2. Refit fifteen times, each time leaving one point out, and report the range of α, β, and E. Which constant is determined worst, and why?
3. The law predicts that the loss difference between two budgets is the same for every model size. Check this against the grid. Which term of Equation 11.20 would have to depend on N to describe what you see?

8. **How the ratio moves with the budget.** ▣ The compute-optimal allocation is given by Equation 11.22.
   1. Show that $D^*/N^* = G^{-2}(C/6)^{(\alpha-\beta)/(\alpha+\beta)}$.
   2. Evaluate D^*/N^* at $C = 10^{21}$ and $C = 10^{25}$ FLOPs for both fits in `fits`. In which direction does the ratio move with the budget under each fit, and by how much?
   3. Both fits come from the same experiments. What do the results say about how precisely a fit of Equation 11.20 determines the token–parameter ratio at budgets a hundred times beyond the largest run that was fitted?

9. **Training for inference.** ▣ A model that serves D_{inf} tokens after training costs about

$$C_{\text{life}} = 6ND + 2ND_{\text{inf}}$$

   FLOPs over its lifetime, since generating a token costs about $2N$ (Section 11.3). Sardana et al. (2024) minimize this cost at a fixed target loss instead of minimizing the loss at a fixed training budget.
   1. Minimize C_{life} subject to $L(N, D) = \ell$ under Equation 11.20. Derive the optimality condition that replaces Equation 11.21, and show that the optimum moves to smaller N and larger D as D_{inf} grows.
   2. Llama 3 8B was pretrained on about 15T tokens. Take the refit of Besiroglu et al. (2024) at face value and find the compute-optimal model with the same predicted loss as $N = 8 \times 10^9$, $D = 1.5 \times 10^{13}$. Compute the ratio of the two training costs and the number of inference tokens at which the two lifetime costs are equal.

10. [extended] **IsoFLOP profiles.** ▣ The sweep of Section 11.7.2.1 fixes three data budgets, so each model receives a different compute budget. Hoffmann et al. (2022) instead fix the budget C and trade N against $D = C/(6N)$.
    1. For each budget $C \in \{3 \times 10^{13}, 10^{14}, 3 \times 10^{14}\}$ FLOPs and each model in `sizes`, train for $C/(6N \cdot 64 \cdot 128)$ steps, rounded, with the recipe of the sweep, and record the validation loss. Plot validation loss against N, one curve per budget.
    2. Take the lowest-loss model of each budget as $N_{\text{opt}}(C)$, and refine it by fitting a parabola in $\log N$ to the points around the minimum. Fit $N_{\text{opt}} \propto C^a$ and compare a with $\beta/(\alpha+\beta)$ for both fits in `fits`.
    3. Count the share of the training text that each run reads. Suppose the corpus had only 5 million bytes, so that the largest runs read it several times. In which direction would repetition bias N_{opt} at the largest budget?

Adapted from Stanford CS336, Assignment 3, Problem (chinchilla_isoflops).

11.8 Further Reading

The sections of this chapter build one block, one language model, one cache, one encoder and one encoder–decoder, one vision model, one mixture of experts, and one small scaling

study. This section points to topics they had no room for: other ways to keep the block stable, the training recipe in more detail, the systems that serve a cache, the uses of each wiring, the recipes that make vision transformers work on modest data, routing beyond the layer of Section 11.6, and the questions that a scaling law leaves open. Several of these topics belong to the Language Models part of the book, and their entries here only say what they change and where they are treated. Implementations, tutorials, and courses for the material that the chapter does teach are listed in its introduction, Chapter 11. Nothing else in the book depends on this section.

11.8.1 The Block

11.8.1.1 Bounds on the Logits

QK-norm, in Section 11.1.2.4, bounds the attention logits by normalizing queries and keys. Released models arrived at it from different starting points: Qwen3 (A. Yang et al. 2025) removed the biases of its query, key, and value projections and added QK-norm, OLMo 2 (Team OLMo et al. 2025) added it together with a loss on the output normalizer, and Gemma 3 (Gemma Team 2025) adopted it in place of the soft-capping of Gemma 2 (Gemma Team 2024). Soft-capping squashes a logit z as $c \tanh(z/c)$, with $c = 50$ for the attention logits and $c = 30$ for the output logits. It bounds the logits without normalizing anything. Fused attention kernels, however, never form the logits, so soft-capping does not fit them. The output-side counterpart is the z-loss of PaLM (Chowdhery et al. 2022), a penalty of about $10^{-4}(\log Z)^2$ on the normalizer Z of the output softmax that keeps it near one. Wortsman et al. (2024) show that small models already exhibit the divergence of output logits that z-loss prevents, once the weight decay is low.

11.8.1.2 Other Placements and No Normalization

Figure 11.2 drew four placements of the normalization. OLMo 2 (Team OLMo et al. 2025) uses a fifth: it normalizes the output of each branch before adding it to the stream, rather than its input. DeepNet (H. Wang et al. 2024) keeps the post-norm arrangement of Section 11.1 but multiplies the stream by a constant larger than one before adding the branch, and scales down the branch's initialization; with these two changes, post-norm stacks of a thousand layers train. Dynamic tanh (Zhu et al. 2025) removes the normalization altogether: it replaces every LayerNorm by an elementwise $\tanh(\alpha x)$ with a learned scalar α, so that no statistic is computed over the token. At the reported scales it matches normalized models in vision and language.

11.8.1.3 Initialization of Deep Stacks

`GPT` in Section 11.2 uses GPT-2's initialization, with the projections that write to the stream scaled by $1/\sqrt{2L}$. SmallInit (Nguyen and Salazar 2019) draws the attention weights with a smaller standard deviation and, together with pre-norm, trains translation models without warmup. OLMo 2 (Team OLMo et al. 2025) went the other way: it dropped its depth-scaled scheme for a plain normal distribution with standard deviation 0.02, and reports that the scales of activations and gradients are better preserved through training. muP (Yang et al. 2022), which Section 9.9 teaches, makes both the initialization and the learning rates depend on width, so that hyperparameters tuned on a small model carry over to a large one.

11.8.1.4 Modifications That Transfer

Many changes to the block were proposed between 2017 and 2020, each with a gain in its own paper. Narang et al. (2021) reimplemented dozens of them in one codebase and found that few improve on the original transformer when the setting changes. RMSNorm and the gated feed-forward networks of Section 11.1 are among the few that do.

11.8.2 The Training Recipe

11.8.2.1 Adam's Second Moment at Small Batch

The runs of Section 11.2 use PyTorch's default $\beta_2 = 0.999$, while reports of large pretraining runs use 0.95. Porian et al. (2024) find that for models of 5M to 220M parameters trained with batches of 128 sequences or fewer, 0.95 is a poor choice and values between 0.99 and 0.999 work better. A small batch gives noisy gradients, and the longer average of a larger β_2 smooths the estimate of their second moment.

11.8.2.2 Constant Rates and Cooldowns

Cosine decay, which `LMOptimizer` uses, needs the length of the run in advance. A constant learning rate followed by a short cooldown matches a tuned cosine schedule, and a cooldown over the last fifth of the steps with a $1 - \sqrt{t}$ shape works best (Hägele et al. 2024). MiniCPM (Hu et al. 2024) named this schedule warmup–stable–decay. For scaling studies this changes the cost: one long run with cooldowns branched off at several points yields the losses of many data budgets, where Section 11.7 trains a separate run for each. Section 9.6 teaches the schedule itself.

11.8.2.3 Pretraining Data

GPT-2 was trained on WebText, about 40 GB of web pages linked from Reddit, and later models train on filtered web crawls of trillions of tokens. How such corpora are filtered, deduplicated, and mixed, and how models are adapted after pretraining, is the subject of the Language Models part.

11.8.3 Generation and the Cache

11.8.3.1 Serving Engines

A server answering many requests at once must decide which sequences share a forward pass and where their caches live. Continuous batching (Yu et al. 2022) schedules at the granularity of one decoding step, admitting and retiring sequences at every step instead of waiting for the longest member of a batch. Paged memory (Kwon et al. 2023) stores each cache in fixed blocks located through a table, like the pages of virtual memory, instead of reserving the maximum length for every sequence as `init_cache` of Section 11.3 does. Prefix caching (Zheng et al. 2024) keeps the caches of finished requests in a tree keyed by their tokens and reuses them when a new request shares a prefix, such as a system prompt. The Language Models part builds these mechanisms; Section 39.7 shows them in use.

11.8.3.2 Speculative Decoding

Decoding reads all weights to produce one token (Section 11.3.2.1). Speculative decoding (Leviathan et al. 2023; C. Chen et al. 2023) lets a small draft model propose several tokens,

which the large model scores in one pass. Let p and q be the distributions of the large and the draft model. Accepting each proposed token with probability $\min(1, p(x)/q(x))$, and resampling a rejected one from the normalized $\max(0, p-q)$, leaves the output distribution exactly that of the large model. Medusa (Cai et al. 2024) replaces the draft model by extra decoding heads, EAGLE (Yuhui Li et al. 2024) drafts from the large model's features, and multi-token prediction (Gloeckle et al. 2024) trains the model to predict several tokens at once. The Language Models part treats these methods.

11.8.3.3 Compressing a Trained Model's Cache

Section 11.3.4.1 compressed GPT-2's cache with a fixed basis and Section 11.3.4.2 with a window. Methods applied after training go further along both lines. Palu (Chang et al. 2025) factorizes the key and value projections of a trained model into low-rank pairs and caches the latent, and Loki (Singhania et al. 2024) keeps the full cache but uses a low-dimensional key subspace to choose the tokens that receive exact attention. H2O (Z. Zhang et al. 2023) keeps the recent tokens plus those that have accumulated the most attention, and SnapKV (Yuhong Li et al. 2024) chooses positions per head from the attention of the last prompt tokens. A benchmark of single questions hides the cost of such evictions. SCBench (Li et al. 2025) finds that methods whose memory grows more slowly than the context fail when later turns of a conversation need what an earlier turn evicted. KVQuant (Hooper et al. 2024) extends the quantization of the section's exercise to three bits per entry, quantizing keys per channel before RoPE and keeping a few outliers exact. The Language Models part treats compression at inference time.

11.8.3.4 Multi-Head Latent Attention in Full

Section 11.3 sketches the latent cache of DeepSeek-V2 (DeepSeek-AI 2024). In full, keys and values of all heads are reconstructed from one compressed vector per token, and the up-projection of the keys can be absorbed into the query projection, so that attention runs on the latent directly. Absorption fails for a rotated key, because the rotation depends on the position and sits between the two matrices, so the model adds a small rotary key shared by all heads beside the latent. DeepSeek-V2 reports a cache 93% smaller than that of its dense 67B predecessor. The Language Models part treats the design in its own section.

11.8.3.5 Gated Attention in Full

Section 11.3.4.2 describes the sigmoid gate of Qiu et al. (2025) as the design that makes sinks unnecessary. The authors compared 30 variants and found the best to be an elementwise gate, computed from the layer input, on each head's output before the output projection. In models of 1.7B dense and 15B mixture-of-experts parameters trained on 3.5T tokens, it lowered perplexity, tolerated larger learning rates, and reduced the attention on the first token from about half to about a twentieth. Qwen3-Next (Qwen Team 2025) uses it.

11.8.4 Wirings and Images

11.8.4.1 Uses of Each Wiring

The three wirings of Section 11.4 divide the work of current systems. Encoders such as ModernBERT (Warner et al. 2024) serve embedding, retrieval, reranking, and classification, where each document is encoded once with context on both sides. Encoder–decoders serve tasks with a distinct input: the T5 family (Raffel et al. 2020) frames every task as text to text,

and Whisper (Radford et al. 2023) decodes text from an encoded recording. Decoders alone serve generation and in-context learning (Brown et al. 2020). Wang et al. (2022) compared the wirings at more than 5B parameters and found the decoder best for zero-shot use after pretraining alone, and an encoder with full visibility of the input best after multitask fine-tuning. T5Gemma (Zhang et al. 2025) converts pretrained decoders into encoder–decoders and finds them stronger at a similar cost of inference. The Language Models part treats pretrained encoders in depth.

11.8.4.2 Learned Queries as an Interface

The Perceiver of Section 11.4 reads a long input through a few learned queries. Vision–language models use the same device to hand images to a language model. Flamingo (Alayrac et al. 2022) compresses a variable number of image and video features into 64 visual tokens, and BLIP-2's Q-Former (Li et al. 2023) is a small transformer of 32 learned queries between a frozen image encoder and a frozen language model. DETR (Carion et al. 2020) used learned object queries earlier, for detection: each of a hundred queries cross-attends into the image features and produces one object or none. Chapter 32 treats detection.

11.8.4.3 Vision Transformers on Modest Data

A vision transformer lacks the locality that a convolution builds in, and Section 11.5 showed what that costs on a small dataset. Several recipes close the gap without more data. DeiT (Touvron, Cord, Douze, et al. 2021) trained on ImageNet-1k alone with strong augmentation and regularization and a distillation token that learns from a convolutional teacher. Beyer et al. (2022) replaced the class token by average pooling and the learned positions by fixed 2-D sinusoids and reached 80% top-1 in 300 epochs, and Steiner et al. (2022) found that with the right augmentation and regularization, more compute gives about the same accuracy as ten times more data. Compact transformers (Hassani et al. 2021) replace the patch stem by a small convolutional tokenizer and the class token by attention pooling; with small patches and 300 epochs, models of under 4M parameters beat ResNet-18 on CIFAR-10.

11.8.4.4 Windows, Two-Dimensional Rotations, and Registers

Swin (Liu et al. 2021) restricts attention to local windows that shift between blocks, and merges patches between stages. This restores locality and a pyramid of resolutions, and makes the cost linear in the number of patches, as detection and segmentation at high resolution require. RoPE in two dimensions (Heo et al. 2024) rotates by the row and the column of a patch and extrapolates to higher resolutions better than a learned table. The register tokens of Section 11.5 need not be trained: Jiang et al. (2025) trace the outlier tokens to a few neurons and move their activation into an extra token at test time, with nearly the effect of trained registers.

11.8.4.5 Self-Supervised and Image–Text Encoders

The ViT of the chapter learns from labels. Masked autoencoders (He et al. 2022) learn by reconstructing masked patches from the quarter of patches that remain, and DINOv2 (Oquab et al. 2024) by self-distillation on a curated image collection. CLIP (Radford et al. 2021) trains an image encoder against a text encoder with a contrastive loss over the batch, and SigLIP (Zhai et al. 2023) replaces the softmax over the batch by a sigmoid loss per pair,

which decouples the loss from the batch size. Encoders of this kind are the vision towers of most vision–language models. Chapter 32 and the Language Models part treat them.

11.8.5 Experts

11.8.5.1 Router Stability

Large sparse models are prone to loss spikes that start in the router, whose logits can grow without bound. ST-MoE (Zoph et al. 2022) adds a router z-loss, a penalty on the squared log-sum-exp of the router logits, which keeps them small without changing which expert wins. With it, ST-MoE trains a sparse model of 269B parameters at the computation per token of a 32B dense model.

11.8.5.2 Expert-Choice Routing

In the layer of Section 11.6 each token chooses its experts, so loads must be balanced afterwards. Expert-choice routing (Zhou et al. 2022) reverses the roles: each expert picks the tokens it scores highest, up to its capacity, so loads are balanced by construction. A token may then be picked by no expert or by many, and its experts depend on the other tokens of the batch, which complicates decoding one token at a time.

11.8.5.3 Sigmoid Gates and Limited Groups

DeepSeek-V3 (DeepSeek-AI, Liu, et al. 2024) scores each expert with its own sigmoid instead of a softmax over experts, normalizes the scores of the selected experts, and restricts the experts of each token to a few groups of devices, which bounds the communication per token. The layer of Section 11.6 keeps the softmax router of the Switch transformer and adopts only DeepSeek-V3's bias controller.

11.8.5.4 Sparsity and Expert Width

Section 11.6 trains one configuration. Clark et al. (2022) fit scaling laws for routed models and find that routing helps at every dense size up to 1.3B parameters, with gains that shrink as the dense model grows. Krajewski et al. (2024) treat the width of the experts as a hyperparameter and scale the tokens with the model; the gap between mixtures and dense models then grows with compute, and experts as wide as the dense feed-forward network are almost never optimal. Later work finds an optimal sparsity that grows with compute (Abnar et al. 2025) and mixtures that beat dense models of the same memory, not only of the same computation (Ludziejewski et al. 2025). Released models have become sparser accordingly: Qwen3-Next (Qwen Team 2025) activates about 3B of its 80B parameters, with 512 experts per layer, and Kimi K2 (Kimi Team 2025a) combines latent attention with 384 experts per layer. The Language Models part treats sparsity in scaling laws.

11.8.5.5 Experts across Devices

When the experts do not fit on one accelerator, different experts live on different devices, and the tokens travel to their experts and back in an all-to-all exchange before and after each expert layer. This is expert parallelism, introduced with GShard (Lepikhin et al. 2021). Fixed capacities, as in the layer of Section 11.6, make the exchange static. The Language Models part treats expert parallelism and the serving of mixtures.

11.8.6 Scaling

11.8.6.1 Reconciling Kaplan and Chinchilla

Kaplan et al. (2020) found that the compute-optimal model size grows as $C^{0.73}$, and Chinchilla (Hoffmann et al. 2022) as $C^{0.50}$. Section 11.7 quotes the finding of Pearce and Song (2024) that counting only non-embedding parameters at small scale explains most of the difference. Porian et al. (2024) account for the rest: the FLOPs of the output layer, the length of the warmup, and optimizer settings tuned to each scale. With these corrected, the two fits agree.

11.8.6.2 Repeated Data

Section 11.7 quotes the finding of Muennighoff et al. (2023) that up to about four passes over the data are nearly as good as fresh data. Their fit gives the whole curve: U unique tokens repeated R times are worth $U + UR^*(1 - e^{-R/R^*})$ fresh tokens, with $R^* \approx 15$, so the value of further passes decays to zero. Budgets beyond the compute optimum are treated in the Language Models part.

11.8.6.3 Precision as a Scaling Variable

Kumar et al. (2025) treat the precision of training as a reduction of the effective parameter count. Quantizing a model after training hurts more the longer it was trained, so that beyond some point more pretraining data makes the quantized model worse. They also find that the compute-optimal precision for training lies around seven to eight bits.

11.8.6.4 Loss and Abilities

Loss falls smoothly with scale, but some abilities measured downstream appear to switch on abruptly as it falls (Wei et al. 2022). Whether such jumps are a property of the models or of metrics that give no credit for partial answers is debated.

11.8.6.5 Departures from the Shared Block

The configurations of Table 11.2 share one block. Three directions depart from it. Hybrids keep full attention in a fraction of the layers and replace the rest with recurrent mixers of linear cost (Kimi Team 2025b), the subject of Chapter 12. Long contexts combine the positional rescaling of Section 10.4, the cache policies of Section 11.3, and systems work; Llama 4 (Meta AI 2025) interleaves attention layers without positional encoding. Learned sparse attention chooses the keys by content. The indexer of DeepSeek-V3.2 (DeepSeek-AI et al. 2025), for example, selects 2,048 key–value tokens per query; Section 10.7 describes it.

Chapter 12

State Space Models

The recurrent networks of Chapter 8 read a text the way we do, one token after another, and carry what they have read in a hidden state. The transformers of Chapter 11 keep every token instead. When a chat assistant answers a question about a long report, it holds a key and a value for every token of the report (Section 11.3), so its memory, and the work for each word of the answer, grow with the length of the report. Can a model read a long text with a memory of fixed size and still model language well?

A recurrent network keeps a summary of fixed size, its *state*, and updates it once per token, so generating the millionth token costs as much as generating the tenth. A state of fixed size raises two problems. Classical recurrent networks update the state nonlinearly, so training processes a sequence one step at a time and leaves most of a GPU idle. And a state of fixed size must forget, which raises the question of what it should keep and how much it can hold. The answers in this chapter come from three directions: gates from the 1990s (Hochreiter and Schmidhuber 1997), linear dynamical systems from physics and control theory (Gu et al. 2020; Gu, Goel, et al. 2022), and attention without the softmax (Katharopoulos et al. 2020). The three meet in recurrent layers that train in parallel and compute a form of attention. Several recent large models interleave such layers with a few attention layers (Lieber et al. 2024; Qwen Team 2025).

Section 12.1 asks how a network can learn what to keep. Section 12.2 asks how to compute the states at all positions at once, and how fast a state should forget. Section 12.3 asks how each input can decide what to write and what to forget. Section 12.4 asks how a state can store key–value pairs, and how many it can hold. Section 12.5 asks how a memory can correct a pair instead of adding to it. Section 12.6 asks what no fixed-size state can do, and how a few attention layers repair it.

We assume the recurrent networks of Chapter 8 and the attention of Chapter 10; linear attention and its computation in chunks (Section 10.5) are the starting point of Section 12.4. The field changes quickly, so the chapter teaches the mechanisms (state, update, readout, and cost) rather than a ranking of current models. It teaches algorithms rather than GPU kernels, which are the subject of Chapter 13, and Section 12.7 points to the topics it has no room for.

Resources and Further Reading

The resources are grouped into implementations to build from, explanations of the ideas, surveys, and course counterparts. All are freely accessible online. Research topics that the chapter does not teach are collected in Section 12.7.

Annotated implementations

- The Annotated S4 — Sasha Rush (2022) and Mamba: The Hard Way (2024) — S4 and the Mamba scan implemented line by line against the papers, in the executable-textbook format this book shares; companions to Section 12.2 and Section 12.3.
- mamba-minimal — the architecture of Section 12.3 in one readable PyTorch file, without the fused kernels.
- state-spaces/mamba — the authors' reference implementation, including the `ssd_minimal` listing that reduces Mamba-2's chunked algorithm (Section 12.4) to a page.
- flash-linear-attention — fla-org — Triton kernels for gated linear attention, DeltaNet, Gated DeltaNet, RWKV, and their relatives; the production form of the chunked layers of Section 12.4 and Section 12.5.
- Gated DeltaNet, from scratch — Sebastian Raschka — a bonus chapter of *Build a Large Language Model (From Scratch)* that implements the Gated DeltaNet layer of Qwen3-Next and Kimi Linear and their three-to-one hybrid ratio; a build-along for Section 12.5 and Section 12.6.
- zoology — HazyResearch — the synthetic associative-recall benchmarks behind the recall experiments of Section 12.6.

The ideas, explained

- A Visual Guide to Mamba and State Space Models — Maarten Grootendorst (2024) — more than fifty figures from the basics of SSMs to the selective scan; a gentle introduction to Section 12.2 and Section 12.3.
- State Space Duality (Mamba-2), parts I–IV — Albert Gu and Tri Dao (2024) — the authors' development of the duality between gated recurrences and masked attention of Section 12.4: model, theory, algorithm, and systems.
- DeltaNet Explained, parts I–III — Songlin Yang (2024) and her Linear Attention and Beyond slides — the delta rule, its chunked training, and the design space of linear attention, from one of its main contributors; a second view of Section 12.5.
- On the Tradeoffs of SSMs and Transformers — Albert Gu (2025) — the argument that compressing a sequence and looking up its tokens are different jobs and that good models need both; the question Section 12.6 measures.
- ASAP seminar series — an ongoing online seminar on sequence-model architectures, where the topics of this chapter continue.

Surveys

- Test-Time Regression — Wang, Shi, and Fox (2025) — the framework behind the regression view of memory at the end of Section 12.5, worked out for attention, linear attention, and many recurrent layers.
- Speed Always Wins — Sun et al. (2025) — a survey of linear sequence models, sparse attention, mixtures of experts, and hybrids, covering the efficient architectures around this chapter.

Courses

- Stanford CS336: Language Modeling from Scratch, Lecture 4 — attention alternatives

and mixtures of experts: this chapter's material as one lecture of a course on building language models.
- CMU 10-423 Generative AI, Lecture 22: State Space Models — lecture slides on S4 and Mamba by Gormley and Virtue; a second treatment of Section 12.2 and Section 12.3.

12.1 Gated Recurrent Networks

The recurrent network of Section 8.5 generates text with real words and punctuation, but its grammar holds over a few words at most. Such networks have trouble remembering anything for long. Consider a model that reads "the keys to the cabinet in the hall" and must predict what comes next. To choose "are" rather than "is", it must remember that the subject, "keys", is plural, and by the time the verb arrives, that word lies 30 characters back. A plain recurrent network rewrites its whole hidden state at every step with a matrix product and a tanh. Information from 30 steps back reaches the present only through 30 of these rewrites, and so does the gradient that would teach the network to keep it. Each rewrite tends to shrink the signal, so plain recurrent networks learn dependencies over a few dozen steps at best, while text, speech, and music are full of longer ones. How can a network keep what it has read for hundreds of steps?

The remedy that made recurrent networks practical is to let the network decide, for each unit and at each step, how much of its state to keep. It dates from the late 1990s, and gated networks carried speech recognition and machine translation through the 2010s, until transformers took their place. A unit that keeps its state unchanged passes the stored information, and the gradient, through the step without loss. The decision is made by a *gate*, a vector of numbers between 0 and 1 that the network computes from its input and its state. We build the two gated networks in wide use, the long short-term memory (LSTM) and the gated recurrent unit (GRU), and test them on a task that needs a long memory and on character-level language modeling. Gates fix the memory, but they read the previous state, so training must still process a sequence one step at a time. The next section removes that restriction.

Throughout the chapter we write the equations for a single sequence, with column vectors; the code processes a minibatch of sequences at once.

```
import os
```

12.1.1 Gates

12.1.1.1 Vanishing Gradients

We first make precise why a plain recurrent network forgets. It updates its hidden state $\mathbf{h}_t \in \mathbb{R}^d$ from the previous state and the input $\mathbf{x}_t$,

$$\mathbf{h}_t = \tanh(\mathbf{W}_{\text{hh}}\mathbf{h}_{t-1} + \mathbf{W}_{\text{xh}}\mathbf{x}_t + \mathbf{b}_{\text{h}}). \tag{12.1}$$

To see how far information travels, we ask how much a small change of the state at step $t-k$ changes the state at step t. One step multiplies a change by the Jacobian $\mathbf{J}_t = \operatorname{diag}(1-\mathbf{h}_t^2)\,\mathbf{W}_{\text{hh}}$, the derivative of the tanh times the recurrent matrix. Over k steps the Jacobians multiply,

$$\frac{\partial \mathbf{h}_t}{\partial \mathbf{h}_{t-k}} = \mathbf{J}_t \mathbf{J}_{t-1} \cdots \mathbf{J}_{t-k+1}, \tag{12.2}$$

and backpropagation sends the gradient of a loss at step t back to step $t-k$ through the transpose of the same product (Section 8.6.2.1). With a single hidden unit the product is a product of numbers $w(1-h_s^2)$, each at most $|w|$ in magnitude. If $|w| < 1$, the product shrinks at least as fast as $|w|^k$: with $w = 0.9$, a change 30 steps back reaches the present at no more than 4% of its size, and a change 100 steps back at no more than 0.003%. If $|w| > 1$, the product can grow until the tanh saturates, and then the factor $1 - h_s^2$ drives it toward zero again. For matrices the same holds with the largest singular value of $\mathbf{W}_{\text{hh}}$ in place of $|w|$. To carry information over hundreds of steps, a plain network would have to keep every factor close to one at every step, by the choice of one matrix shared by all steps. Gradient clipping limits gradients that grow; nothing recovers a gradient that has vanished.

12.1.1.2 An Additive Memory

The simplest state whose Jacobian is exactly one is a running sum, $c_t = c_{t-1} + x_t$. Its derivative $\partial c_t / \partial c_{t-k}$ is 1 for every k: an input written at step 1 is still in the sum at step 1,000, and the gradient reaches it undiminished. But a running sum cannot forget. Every input stays in it forever, so after many steps it holds a mixture of everything the network has read, and its size grows with the length of the sequence.

12.1.1.3 Gates as Learned Decay

A useful memory keeps some things and drops others. We let the network choose, at every step, a fraction $f_t \in (0,1)$ of the old state to keep and a fraction $i_t \in (0,1)$ of new content $\tilde{c}_t$ to write:

$$c_t = f_t \, c_{t-1} + i_t \, \tilde{c}_t. \tag{12.3}$$

Along this path the derivative over k steps is the product of the kept fractions, $\partial c_t / \partial c_{t-k} = f_t f_{t-1} \cdots f_{t-k+1}$. If the network keeps a constant fraction f, a stored value decays like f^k and halves every $\ln 2 / \ln(1/f)$ steps, its *half-life*: about 7 steps for $f = 0.9$, 69 for $f = 0.99$, and 693 for $f = 0.999$. A network that sets f_t to one holds a value, and its gradient, for as long as it likes; setting f_t to zero erases it.

The fractions are computed by *gates*. A gate is a layer with a sigmoid output, one value in $(0,1)$ per unit, that reads the current input and the previous output of the network. For the fraction kept, one per unit,

$$\mathbf{f}_t = \sigma(\mathbf{W}_{\text{xf}} \mathbf{x}_t + \mathbf{W}_{\text{hf}} \mathbf{h}_{t-1} + \mathbf{b}_{\text{f}}). \tag{12.4}$$

A gate multiplies a signal elementwise. A value near 1 lets the signal pass, a value near 0 blocks it. Because the gate reads $\mathbf{x}_t$ and $\mathbf{h}_{t-1}$, the decision depends on what the network is reading and what it has stored; because σ is smooth, the decision is learned by gradient descent like every other weight.

Compare this with the plain network. There, the whole state is multiplied by the matrix $\mathbf{W}_{\text{hh}}$, the same at every step, and pushed through a tanh. In the gated memory, each unit

is multiplied by its own number, chosen anew at every step, and nothing else happens to it. A product of numbers below one cannot grow, so the memory path cannot explode, and a gate held near one keeps it from vanishing. One detail matters in practice: with small weights, a gate starts near $\sigma(0) = 1/2$, a half-life of one step. A gated network can learn long memories, but it does not start with them.

12.1.2 Long Short-Term Memory

12.1.2.1 The Memory Cell

The *long short-term memory* (LSTM) of Hochreiter and Schmidhuber (1997), with the forget gate added by Gers et al. (2000), builds a recurrent layer around the gated sum Equation 12.3. Its name contrasts two kinds of memory: the weights of a network change slowly and store what holds for all sequences, while its activations are overwritten at every step. The LSTM's *memory cell* $\mathbf{c}_t \in \mathbb{R}^d$ is an activation designed to persist. Three gates of the form Equation 12.4 control it: the *input gate* $\mathbf{i}_t$, the *forget gate* $\mathbf{f}_t$, and the *output gate* $\mathbf{o}_t$,

$$\begin{aligned} \mathbf{i}_t &= \sigma(\mathbf{W}_{\mathrm{xi}}\mathbf{x}_t + \mathbf{W}_{\mathrm{hi}}\mathbf{h}_{t-1} + \mathbf{b}_{\mathrm{i}}), \\ \mathbf{f}_t &= \sigma(\mathbf{W}_{\mathrm{xf}}\mathbf{x}_t + \mathbf{W}_{\mathrm{hf}}\mathbf{h}_{t-1} + \mathbf{b}_{\mathrm{f}}), \\ \mathbf{o}_t &= \sigma(\mathbf{W}_{\mathrm{xo}}\mathbf{x}_t + \mathbf{W}_{\mathrm{ho}}\mathbf{h}_{t-1} + \mathbf{b}_{\mathrm{o}}). \end{aligned} \tag{12.5}$$

A fourth layer of the same form, with a tanh in place of the sigmoid so that content can be negative, proposes what to write, the *input node* $\tilde{\mathbf{c}}_t = \tanh(\mathbf{W}_{\mathrm{xc}}\mathbf{x}_t + \mathbf{W}_{\mathrm{hc}}\mathbf{h}_{t-1} + \mathbf{b}_{\mathrm{c}})$. The cell is the gated sum Equation 12.3, one copy per unit, and the output gate decides how much of the cell the rest of the network sees:

$$\mathbf{c}_t = \mathbf{f}_t \odot \mathbf{c}_{t-1} + \mathbf{i}_t \odot \tilde{\mathbf{c}}_t, \qquad \mathbf{h}_t = \mathbf{o}_t \odot \tanh(\mathbf{c}_t). \tag{12.6}$$

Here $\odot$ is the elementwise product. The cell is the network's memory, and $\mathbf{h}_t$ is what it reports: the next layer, the output layer, and the gates of the next step read $\mathbf{h}_t$, never $\mathbf{c}_t$ directly. The output gate lets a unit hold a value without showing it, for instance a plural subject that matters only when a verb arrives, and reveal it when it becomes relevant. Figure 12.1 shows the data flow.

Follow the cell along the top of the figure. Between $\mathbf{c}_{t-1}$ and $\mathbf{c}_t$ there is one elementwise product and one sum, no matrix and no saturating nonlinearity, so with the gates held fixed the Jacobian $\partial\mathbf{c}_t/\partial\mathbf{c}_{t-1}$ is the diagonal matrix $\mathrm{diag}(\mathbf{f}_t)$. A reader may object that the gates are not fixed: they read $\mathbf{h}_{t-1}$, which depends on $\mathbf{c}_{t-1}$, so the full derivative contains further terms that pass through the gates and the recurrent matrices, with the products of Equation 12.2. That is correct. The cell does not guarantee that every gradient survives; it provides one path along which gradients can survive, and training decides whether the gates keep it open.

12.1.2.2 Implementation

We write the layer out step by step. The four layers that compute $\mathbf{i}_t$, $\mathbf{f}_t$, $\tilde{\mathbf{c}}_t$, and $\mathbf{o}_t$ read the same inputs, so we stack their weights and compute all four with one matrix product per input, then split the result. In the code each row of a minibatch is one sequence, so the weights multiply from the right and hold the transposes of the matrices in the equations.

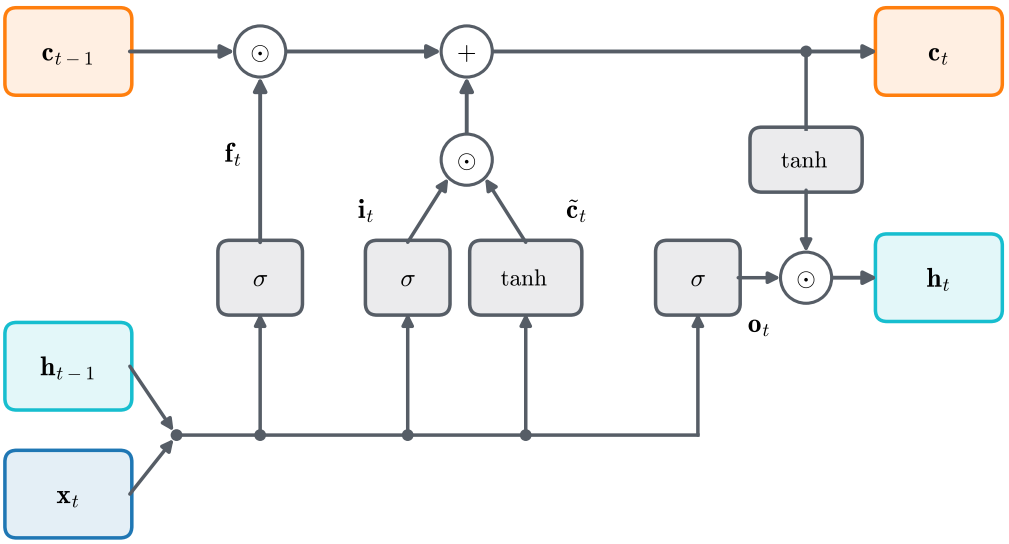


Figure 12.1: The LSTM memory cell. The previous output $\mathbf{h}_{t-1}$ and the input $\mathbf{x}_t$ feed four layers: three sigmoid gates and a tanh input node. The cell $\mathbf{c}_t$ is updated additively, scaled by the forget gate, and the output gate controls what the rest of the network sees.

```
class LSTMScratch(nn.Module):
    """An LSTM layer computed one time step at a time."""
    def __init__(self, num_inputs, num_hiddens):
        super().__init__()
        self.num_hiddens = num_hiddens
        scale = 1 / math.sqrt(num_hiddens)
        # Weights of i, f, c_tilde, and o, stacked along the last axis
        self.W_x = nn.Parameter(torch.randn(num_inputs, 4 * num_hiddens) * scale)
        self.W_h = nn.Parameter(torch.randn(num_hiddens, 4 * num_hiddens) * scale)
        self.b = nn.Parameter(torch.zeros(4 * num_hiddens))

    def forward(self, X):  # X: (batch_size, num_steps, num_inputs)
        h = c = torch.zeros(X.shape[0], self.num_hiddens, device=X.device)
        outputs = []
        for x in X.unbind(1):  # One time step at a time
            i, f, c_tilde, o = (x @ self.W_x + h @ self.W_h + self.b).chunk(4, -1)
            c = torch.sigmoid(f) * c + torch.sigmoid(i) * torch.tanh(c_tilde)
            h = torch.sigmoid(o) * torch.tanh(c)
            outputs.append(h)
        return torch.stack(outputs, 1)  # (batch_size, num_steps, num_hiddens)
```

Deep learning frameworks provide the LSTM as a library layer that runs the same recurrence in a fused, compiled loop. To check that it computes Equation 12.5 and Equation 12.6 and nothing else, we copy its weights into our layer and compare the outputs on random inputs.

```
lstm = nn.LSTM(16, 32, batch_first=True)  # Stacks its weights as i, f, c~, o
cell = LSTMScratch(16, 32)
with torch.no_grad():
    cell.W_x.copy_(lstm.weight_ih_l0.T)
    cell.W_h.copy_(lstm.weight_hh_l0.T)
    cell.b.copy_(lstm.bias_ih_l0 + lstm.bias_hh_l0)
X = torch.randn(4, 9, 16)  # (batch_size, num_steps, num_inputs)
print(f'max difference: {(cell(X) - lstm(X)[0]).abs().max():.1e}')
```

```
max difference: 8.9e-08
```

The outputs agree to the rounding error of 32-bit arithmetic. From now on we use the library layer, which is much faster. It is still a loop over time steps, each waiting for the one before.

12.1.3 Gated Recurrent Units

The LSTM keeps two vectors per step, the cell and the output, and computes four gated layers. The *gated recurrent unit* (GRU) of Cho, Van Merriënboer, Bahdanau, et al. (2014) keeps one vector and computes three, and it trains about as well as the LSTM on many tasks (Chung et al. 2014).

12.1.3.1 Reset and Update Gates

The GRU has two gates of the form Equation 12.4, a *reset gate* $\mathbf{r}_t$ and an *update gate* $\mathbf{z}_t$:

$$\mathbf{r}_t = \sigma(\mathbf{W}_{\text{xr}}\mathbf{x}_t + \mathbf{W}_{\text{hr}}\mathbf{h}_{t-1} + \mathbf{b}_{\text{r}}), \qquad \mathbf{z}_t = \sigma(\mathbf{W}_{\text{xz}}\mathbf{x}_t + \mathbf{W}_{\text{hz}}\mathbf{h}_{t-1} + \mathbf{b}_{\text{z}}). \tag{12.7}$$

The reset gate acts inside a plain recurrent update that proposes a *candidate state*,

$$\tilde{\mathbf{h}}_t = \tanh\left(\mathbf{W}_{\text{xh}}\mathbf{x}_t + \mathbf{W}_{\text{hh}}(\mathbf{r}_t \odot \mathbf{h}_{t-1}) + \mathbf{b}_{\text{h}}\right). \tag{12.8}$$

With $\mathbf{r}_t$ near one the candidate is the plain recurrent update; with $\mathbf{r}_t$ near zero it ignores the state and reads only the current input, as if the sequence started anew. The update gate then blends the old state with the candidate:

$$\mathbf{h}_t = \mathbf{z}_t \odot \mathbf{h}_{t-1} + (1 - \mathbf{z}_t) \odot \tilde{\mathbf{h}}_t. \tag{12.9}$$

This is the gated sum Equation 12.3 with the input gate tied to the forget gate, $i_t = 1 - f_t$: whatever fraction of the old state is dropped is filled with new content. Where $\mathbf{z}_t$ is near one, the state is copied and the step is skipped; where it is near zero, the state is overwritten. Each coordinate of $\mathbf{h}_t$ is a weighted average of two numbers in $[-1, 1]$, so the state stays bounded without a second tanh, and one vector serves as both memory and output. Figure 12.2 shows the cell. With input and hidden width d, a GRU layer has $3 \cdot 2d^2$ weights against the LSTM's $4 \cdot 2d^2$.

Writing the GRU from scratch would repeat `LSTMScratch` with three gated layers in place of four; an exercise asks for it.

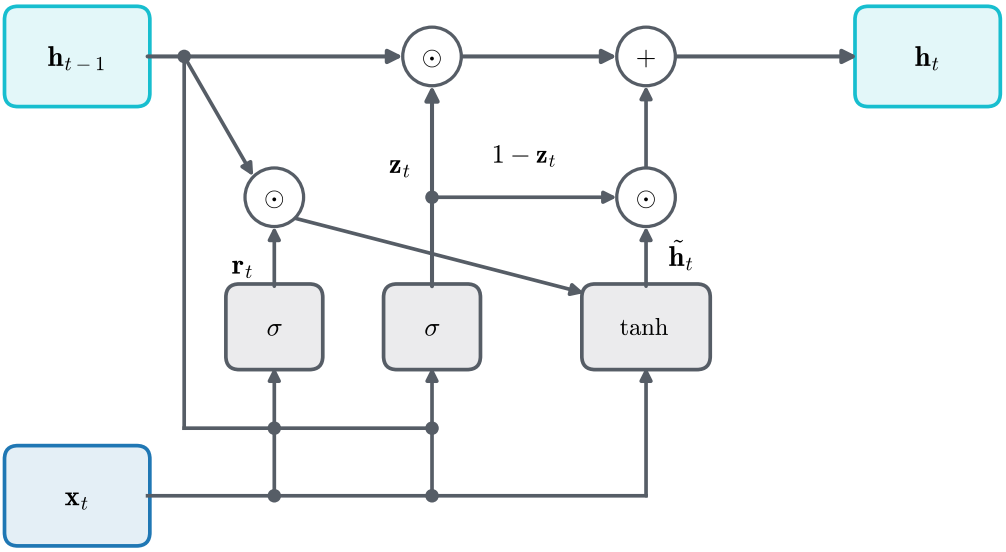


Figure 12.2: The GRU cell. The reset gate filters the previous state inside the candidate $\tilde{\mathbf{h}}_t$; the update gate blends the old state and the candidate.

12.1.4 Training Gated Networks

12.1.4.1 Delayed Recall

How long can each network keep a token? We test it directly with a *delayed recall* task. A sequence starts with one of eight symbols, followed by $T - 1$ filler tokens drawn at random from eight other symbols, and at the end the network must name the first symbol. It can do so only by carrying the symbol through $T - 1$ updates, each of which reads a token that does not matter. Chance accuracy is 1/8.

```
def delayed_recall(rng, batch_size, num_steps, num_symbols=8,
                   blank_fillers=False):
    """A symbol, then num_steps - 1 fillers; the target is the symbol."""
    Y = rng.integers(0, num_symbols, (batch_size, 1))  # One query, at the end
    X = rng.integers(num_symbols, 2 * num_symbols, (batch_size, num_steps))
    if blank_fillers:  # The same filler token at every step
        X[:] = num_symbols
    X[:, :1] = Y
    return X, Y

X, Y = delayed_recall(np.random.default_rng(0), 2, 6)
print(X, Y)
```

```
[[ 6 10 10  8  8  8]
 [ 5 14 13 15 12 12]] [[6]
 [5]]
```

The model embeds the tokens in 32 dimensions, runs one recurrent layer with 64 hidden

units, and applies a classifier to the layer's output at every position. The answer is read at the last position; the tasks of later sections read it at several.

The chapter's models initialize their own linear and convolutional layers with `init_lecun`. It draws the weights from the LeCun normal distribution, of variance $1/n_{\text{in}}$ (Section 4.4), and sets the biases to zero. A projection then passes on the variance of its input, so signals keep their scale through the projections, gates, and residual blocks of these models. The default of `nn.Linear` draws weights and biases uniformly from $[-1/\sqrt{n_{\text{in}}}, 1/\sqrt{n_{\text{in}}}]$, a weight variance of $1/(3n_{\text{in}})$; with it, whether gated linear attention learns the recall task of Section 12.6 depends on the random seed. Biases with a purpose, such as the gate biases of later sections, are set after `init_lecun`. It also covers the library's recurrent layers, whose default draws all weights and biases from the same uniform distribution with n_{in} replaced by the hidden size: their input and recurrent matrices get LeCun normal weights, and their biases start at zero.

```
def init_lecun(module):
    """LeCun normal weights and zero biases for all linear, convolutional,
    and recurrent layers."""
    for layer in module.modules():
        if isinstance(layer, (nn.Linear, nn.Conv1d)):
            fan_in = layer.weight[0].numel()  # Inputs of one output unit
            nn.init.normal_(layer.weight, std=fan_in ** -0.5)
            if layer.bias is not None:
                nn.init.zeros_(layer.bias)
        elif isinstance(layer, nn.RNNBase):  # Input and recurrent matrices
            for name, param in layer.named_parameters():
                if name.startswith('weight'):
                    nn.init.normal_(param, std=param.shape[1] ** -0.5)
                else:
                    nn.init.zeros_(param)
    return module
```

The library's recurrent layers return their outputs together with their final state; a small wrapper keeps only the outputs, which is the interface that all sequence layers of this chapter share.

```
class Recurrent(nn.Module):
    """A batch-first recurrent layer that returns only its outputs."""
    def __init__(self, rnn):
        super().__init__()
        self.rnn = rnn

    def forward(self, X):
        return self.rnn(X)[0]

class RecallModel(nn.Module):
    """Embedding, a sequence layer, and a classifier at every position."""
    def __init__(self, layer, num_hiddens, vocab_size=16, num_inputs=32,
                 num_outputs=8):
        super().__init__()
        self.embedding = nn.Embedding(vocab_size, num_inputs)
```

```
        nn.init.normal_(self.embedding.weight, std=num_inputs ** -0.5)
        self.layer = layer
        self.head = init_lecun(nn.Linear(num_hiddens, num_outputs))

    def forward(self, X):  # Logits, (batch_size, num_steps, num_outputs)
        return self.head(self.layer(self.embedding(X)))
```

Besides the three networks with their gates starting near one half, we train an LSTM and a GRU whose gates start out keeping their state: the bias of the LSTM's forget gate is set to 5 and that of its input gate to -5, and the bias of the GRU's update gate to 5. Since $\sigma(5) = 0.993$, these gates start with a half-life of about 100 steps instead of one.

```
def init_keep_gate(rnn, bias=5.0):
    """Start an LSTM's forget gate, or a GRU's update gate, near one."""
    h = rnn.hidden_size
    with torch.no_grad():
        rnn.bias_ih_l0.zero_()  # Independent of the earlier initialization
        rnn.bias_hh_l0.zero_()
        rnn.bias_ih_l0[h:2 * h] = bias  # LSTM: i, f, c~, o; GRU: r, z, n
        if isinstance(rnn, nn.LSTM):
            rnn.bias_ih_l0[:h] = -bias  # Input gate near zero
    return rnn
```

The synthetic tasks of this chapter are all trained the same way, so we save the loop once. `train_queries` draws a fresh batch from the function `batch` at every step, computes the cross-entropy at the query positions only, the last `Y.shape[1]` positions of the sequence, and takes an optimizer step with the gradient clipped to norm 1. `query_accuracy` returns the fraction of queries answered correctly.

```
def train_queries(model, batch, num_steps, optimizer,
                  scheduler=None):
    """Train on fresh batches; the queries are the last Y.shape[1] steps."""
    device = next(model.parameters()).device
    for step in range(num_steps):
        X, Y = (torch.as_tensor(A, device=device) for A in batch(step))
        logits = model(X)[:, -Y.shape[1]:]  # Predictions at the queries
        loss = F.cross_entropy(logits.flatten(0, 1), Y.flatten())
        optimizer.zero_grad()
        loss.backward()
        nn.utils.clip_grad_norm_(model.parameters(), 1.0)
        optimizer.step()
        if scheduler is not None:
            scheduler.step()

def query_accuracy(model, X, Y):
    """Fraction of queries answered correctly."""
    device = next(model.parameters()).device
    X, Y = (torch.as_tensor(A, device=device) for A in (X, Y))
    with torch.inference_mode():
        return (model(X)[:, -Y.shape[1]:].argmax(-1) == Y).float().mean()
```

For delayed recall, each model is trained with Adam on 1,500 batches of 128 sequences and tested on 2,048 new sequences.

```
def train_delayed_recall(model, num_steps, blank_fillers=False,
                         num_updates=1500, lr=3e-3):
    """Train on fresh batches of delayed recall; return the test accuracy."""
    task = lambda rng, n: delayed_recall(rng, n, num_steps,
                                         blank_fillers=blank_fillers)
    rng = np.random.default_rng(0)
    model.to(d2l.try_gpu())
    train_queries(model, lambda step: task(rng, 128), num_updates,
                  torch.optim.Adam(model.parameters(), lr=lr))
    return float(query_accuracy(model, *task(np.random.default_rng(1), 2048)))
```

```
def recurrent(layer, keep_gate=False):
    """A batch-first layer, 32 inputs to 64 hidden units, LeCun-initialized."""
    layer = init_lecun(layer(32, 64, batch_first=True))
    return init_keep_gate(layer) if keep_gate else layer

layers = {'RNN': lambda: recurrent(nn.RNN), 'LSTM': lambda: recurrent(nn.LSTM),
          'GRU': lambda: recurrent(nn.GRU),
          'LSTM, keep gate': lambda: recurrent(nn.LSTM, keep_gate=True),
          'GRU, keep gate': lambda: recurrent(nn.GRU, keep_gate=True)}
lengths = [5, 10, 20, 50, 100, 200]
print(' ' * 16, ' '.join(f'{T:5d}' for T in lengths))
for name, layer in layers.items():
    accuracy = [train_delayed_recall(RecallModel(Recurrent(layer()), 64), T)
                for T in lengths]
    print(f'{name:16s}', ' '.join(f'{a:5.2f}' for a in accuracy))
```

```
                    5    10    20    50   100   200
RNN              1.00  1.00  1.00  0.50  1.00  0.13
LSTM             1.00  1.00  0.37  0.12  0.12  0.12
GRU              1.00  1.00  1.00  0.12  0.13  0.12
LSTM, keep gate  1.00  1.00  1.00  1.00  1.00  1.00
GRU, keep gate   1.00  1.00  1.00  1.00  1.00  1.00
```

With their gates starting near one half, the LSTM and the GRU recall the symbol after 10 steps and are at chance after 50; in some runs the LSTM recalls only part of the symbols already at 20. The plain RNN is erratic: from 50 steps on, it recalls anywhere from none to all of the symbols, and which it is changes from run to run and from one delay to the next. The LSTM and the GRU whose keep gates start near one recall it after 200 steps. The architecture is the same in both pairs of rows; only the initialization differs. A gate that starts at one half halves the memory at every step, so the gradient that would teach it to keep the symbol shrinks like 2^{-k} over k steps, and training never finds the setting that would solve the task. A gate that starts near one passes the gradient back through hundreds of steps. Gates make a long memory possible and learnable, but whether training finds it depends on where the gates start.

12.1.4.2 Deep Recurrent Networks

A single recurrent layer computes one nonlinear update per step. To compute more complex functions of the past, we stack layers: the outputs of one layer are the input sequence of the next. Stacked recurrent layers are trained with the same techniques as stacked transformer blocks (Section 11.1): each layer is wrapped in a *residual connection*, which adds the layer's output to its input, with a layer normalization in front, and is followed by a small multilayer perceptron (MLP) that transforms each position separately. The residual connections give the gradient a direct path through the depth of the network, as the gated cell gives it one through time.

We save this stack once, because every model in the chapter is an instance of it. `SeqLM` embeds the tokens, applies one residual block per *sequence mixer*, our name for the layer that passes information along the sequence, and predicts the next token from each position. The mixers are passed in as a list, so the same stack can hold an LSTM here, the layers of the coming sections, or a mixture of them. Each mixer maps a minibatch of sequences of shape (batch size, number of steps, d) to one of the same shape. The token embeddings start with entries of standard deviation $1/\sqrt{d}$, so that every embedded token has about unit length.

```
class SeqLM(nn.Module):
    """A language model of residual blocks: a sequence mixer, then an MLP."""
    def __init__(self, vocab_size, num_hiddens, mixers):
        super().__init__()
        d = num_hiddens
        self.embedding = nn.Embedding(vocab_size, d)
        nn.init.normal_(self.embedding.weight, std=d ** -0.5)  # Unit length
        self.mixers = nn.ModuleList(mixers)
        self.norms = nn.ModuleList([nn.LayerNorm(d) for _ in mixers])
        self.mlps = nn.ModuleList([nn.Sequential(
            nn.LayerNorm(d), nn.Linear(d, 4 * d), nn.GELU(),
            nn.Linear(4 * d, d)) for _ in mixers])
        self.norm = nn.LayerNorm(d)
        self.head = nn.Linear(d, vocab_size)
        for module in (self.mlps, self.head):  # Mixers initialize themselves
            init_lecun(module)

    def forward(self, X):
        H = self.embedding(X)  # (batch_size, num_steps, num_hiddens)
        for norm, mixer, mlp in zip(self.norms, self.mixers, self.mlps):
            H = H + mixer(norm(H))
            H = H + mlp(H)
        return self.head(self.norm(H))  # (batch_size, num_steps, vocab_size)
```

A recurrent layer can also read the sequence backward. A *bidirectional* network (Schuster and Paliwal 1997) runs a second layer from the last step to the first and concatenates the two outputs, so that the representation of each position depends on the whole sequence. This helps whenever the whole input is given, as in tagging the words of a sentence or encoding a sentence for translation. It cannot be used for language modeling: the backward layer reads the token that the model is asked to predict, so the training loss falls without the model learning to predict anything, and at generation time the future tokens do not exist.

12.1.4.3 Character-Level Language Modeling

Every section of this chapter trains its models on the same language modeling task, so that their perplexities can be compared. The text is the *Penn Treebank* (Marcus et al. 1993), about one million words of *Wall Street Journal* articles, in the version commonly used for language modeling: lowercased, with numbers replaced by `N`, rare words by `<unk>`, and a standard split into training and validation text. We model it character by character, with a vocabulary of about 50 characters, and cut the text into windows of 128 characters, which cover about 25 words. The training text has 5 million characters. *The Time Machine* of Section 8.2, with under 200,000, is too small for the models of this chapter: they memorize it within a few thousand training steps.

```
class PennTreebank(d2l.DataModule):
    """Character-level Penn Treebank, cut into windows of num_steps."""
    def __init__(self, batch_size, num_steps):
        super().__init__()
        self.save_hyperparameters()
        data_dir = d2l.download_extract('ptb')
        def read(split):
            with open(os.path.join(data_dir, f'ptb.{split}.txt')) as f:
                return f.read()
        train, val = read('train'), read('valid')
        self.vocab = d2l.Vocab(list(train))
        self.train, self.val = self.windows(train), self.windows(val)

    def windows(self, text):
        corpus = d2l.tensor(self.vocab[list(text)])
        n = (len(corpus) - 1) // self.num_steps * self.num_steps
        return (corpus[:n].reshape(-1, self.num_steps),  # Inputs
                corpus[1:n + 1].reshape(-1, self.num_steps))  # Next tokens

    def get_dataloader(self, train):
        return self.get_tensorloader(self.train if train else self.val, train)
```

```
data = PennTreebank(batch_size=64, num_steps=128)
print(f'{len(data.vocab)} characters, {len(data.train[0])} training and '
      f'{len(data.val[0])} validation windows')
```

```
51 characters, 39856 training and 3123 validation windows
```

We measure a model by its *perplexity* on the validation text, the exponential of the average cross-entropy loss per character (Section 8.3). A perplexity of 3 means that the model is, on average, as uncertain about the next character as if it chose uniformly among 3 candidates. `d2l.lm_perplexity` of Section 9.5 computes it over the validation minibatches, with dropout off.

All models are trained with the same recipe: AdamW (Section 9.5.4) with learning rate 0.002 and weight decay 0.01, for 3,000 steps of 64 windows, using the training loop `d2l.train_steps` of Section 9.5, which draws minibatches and takes optimizer steps. The 3,000 steps cover the training text about five times. The models do not overfit in that time: the validation perplexity of every model in this chapter is still falling at the last step, so the comparisons are at a fixed budget, not at convergence.

```
def fit_lm(model, data, num_steps=3000, lr=2e-3):
    """Train with AdamW for num_steps steps; return validation perplexity."""
    optimizer = torch.optim.AdamW(model.parameters(), lr=lr,
                                  weight_decay=0.01)
    d2l.train_steps(model, data, optimizer, num_steps)
    return d2l.lm_perplexity(model, data.val_dataloader())
```

We train three models of width $d = 128$ with two blocks each, whose mixers are a plain recurrent layer, an LSTM, and a GRU, and count their trainable parameters with `d2l.num_params` of Section 7.5.

```
results = {}
for name, layer in {'RNN': nn.RNN, 'LSTM': nn.LSTM, 'GRU': nn.GRU}.items():
    mixers = [Recurrent(init_lecun(layer(128, 128, batch_first=True)))
              for _ in range(2)]
    model = SeqLM(len(data.vocab), 128, mixers)
    results[name] = d2l.num_params(model), fit_lm(model, data)
```

```
for name, (num_params, ppl) in results.items():
    print(f'{name:5s} {num_params:9,d} parameters   perplexity {ppl:.2f}')
```

```
RNN     343,859 parameters   perplexity 3.13
LSTM    542,003 parameters   perplexity 2.97
GRU     475,955 parameters   perplexity 3.00
```

All three models reach validation perplexities between about 2.95 and 3.15. The gated networks are a few percent better than the plain one, and the LSTM and the GRU differ by one or two percent. Character-level language modeling at this scale rewards long memory little: the next character of a word depends mostly on the last few characters, which a plain recurrent network keeps without difficulty. The delayed recall task isolates long memory, and on it the models differ sharply. Later sections report both kinds of result, and the language modeling task serves mainly to check that a new model learns ordinary text as well as the old ones.

The gates solved the memory problem, but not the speed problem. Every gate reads $\mathbf{h}_{t-1}$, so step t cannot begin until step $t-1$ has finished, and a sequence of T tokens takes T dependent steps to train, however many processors are available. The next section asks what remains of a gated network when the gates read only the input.

12.1.5 Summary

A plain recurrent network passes information, and gradients, from step to step through a product of Jacobians, which shrinks or grows geometrically with the distance, so it learns dependencies over a few dozen steps at best. Gated networks add a memory that each unit updates as a gated sum: it keeps a fraction of its old value and adds a fraction of new content, both fractions computed by sigmoid gates from the current input and the previous output. Along this path the gradient is a product of the kept fractions, which the network can hold near one. The LSTM keeps a separate cell controlled by input, forget, and output gates; the GRU blends its old state with a candidate through one update gate and filters the candidate

with a reset gate. Both remember over spans where a plain network fails, and both still train one step at a time, because their gates read the previous state.

12.1.6 Exercises

1. **Holding a value.** Suppose that the character at step s determines the prediction at a much later step t, and that one LSTM unit is to carry it from s to t.
   1. State the value that each gate of that unit in Equation 12.5 should take at step s, at the steps strictly between s and t, and at step t. Then do the same for the update gate of a GRU unit in Equation 12.9.
   2. A sigmoid gate never reaches 1. Suppose that the forget gate's pre-activation stays at a constant b, so that its value is $\sigma(b)$ at every step. What fraction of the stored value survives k steps along the direct path? How large must b be for 90% of the value to survive 1,000 steps?
   3. Show that a forget-gate bias of $\log(T-1)$ gives an initial forget gate of $1-1/T$ when the weights are zero, so that for large T a stored value decays by a factor of about e over T steps. The *chrono initialization* of Tallec and Ollivier (2018) draws the forget-gate bias of each unit as $\log u$, with u uniform on $[1, T_{\text{max}}-1]$, and sets $\mathbf{b}_{\text{i}} = -\mathbf{b}_{\text{f}}$. Show that with zero weights this makes $\mathbf{f}_t + \mathbf{i}_t = 1$, and compare the resulting cell update with Equation 12.9.
2. **Output squashing.** The input node $\tilde{\mathbf{c}}_t$ is already squashed by a tanh, yet Equation 12.6 applies tanh to $\mathbf{c}_t$ again before the output gate.
   1. Starting from $\mathbf{c}_0 = \mathbf{0}$, show that every entry of $\mathbf{c}_t$ is smaller than t in magnitude. Show further that if every forget-gate value is at most $f_{\text{max}} < 1$, every entry stays below $1/(1-f_{\text{max}})$ in magnitude, whatever t is.
   2. Suppose that the output were $\mathbf{h}_t = \mathbf{o}_t \odot \mathbf{c}_t$ instead. After many steps of accumulation, what happens to the gate pre-activations at the next step, and what does that do to the gradients that flow through the gates?
3. code **Forget-gate bias.** With small weights, every forget gate starts near $\sigma(0) = 1/2$. Jozefowicz et al. (2015) found that a forget-gate bias of 1 let the LSTM nearly match the GRU on their benchmarks.
   1. In PyTorch, the forget-gate bias of `nn.LSTM` is the second quarter of `bias_ih_l0` (and of `bias_hh_l0`); in Flax it is the bias of the layer `hf` of `nnx.LSTMCell`. Initialize it to 1, repeat the delayed recall experiment of Section 12.1.4.1, and compare the longest delay at which the LSTM succeeds with and without the bias. Repeat both with a second random seed.
   2. Use the chrono initialization of the previous exercise with T_{max} equal to the delay. Does it extend the range further?
4. **Reset before or after.** The fused GRU layers of deep learning frameworks apply the reset gate after the recurrent matrix product. Apart from the placement of the bias terms, they compute the candidate

$$\tilde{\mathbf{h}}_t = \tanh(\mathbf{W}_{\text{xh}}\mathbf{x}_t + \mathbf{r}_t \odot (\mathbf{W}_{\text{hh}}\mathbf{h}_{t-1}) + \mathbf{b}_{\text{h}})$$

   in place of Equation 12.8.
   1. Show that the two candidates agree for every $\mathbf{r}_t$ and $\mathbf{h}_{t-1}$ if and only if $\mathbf{W}_{\text{hh}}$ is diagonal. For a general $\mathbf{W}_{\text{hh}}$, which reset-gate values still make them agree?
   2. The input products $\mathbf{W}_{\text{x}\cdot}\mathbf{x}_t$ of all steps can be computed as one large product before the recurrence starts. Under each form of the candidate, how many matrix products involving the previous state must each step perform one after the other? Why do fused GRU kernels use the form above?

5. code **GRU from scratch.** Write a class `GRUScratch` by editing `LSTMScratch`: three heads following Equation 12.7 and Equation 12.8, the update Equation 12.9, and a state that consists of $\mathbf{h}_t$ alone.
   1. Copy the weights of the library GRU layer into your layer, as we did for the LSTM, and compare the outputs. They will not agree. Use the previous exercise to change one line of your layer so that they do.
   2. Remove the reset gate by fixing $\mathbf{r}_t = \mathbf{1}$, and in a second layer remove the update gate by fixing $\mathbf{z}_t = \mathbf{0}$. Before training, predict from Equation 12.9 which removal hurts delayed recall more. Then run the experiment of Section 12.1.4.1 and check your prediction.
6. code **The cell's other paths.** ▣ Consider an LSTM with one input and one hidden unit, so that every quantity in Equation 12.5 and Equation 12.6 is a scalar, and write w_{hf}, w_{hi}, and w_{hc} for the recurrent weights of the forget gate, the input gate, and the input node.
   1. The output gate o_{t-1} depends on h_{t-2} but not on c_{t-1}. Using $h_{t-1} = o_{t-1}\tanh(c_{t-1})$, derive the total derivative $\mathrm{d}c_t/\mathrm{d}c_{t-1}$. Write it as the direct-path value f_t plus a correction, and factor the correction into a term from step t and the derivative $\mathrm{d}h_{t-1}/\mathrm{d}c_{t-1}$.
   2. For which values of the gates, of $\tilde{c}_t$, and of c_{t-1} does the correction vanish? Which of these conditions can the network set through its gates while the cell is storing a value?
   3. Train an LSTM on delayed recall at a delay where it succeeds. For one test sequence, compute the norm of the gradient of the loss with respect to the cell $\mathbf{c}_{T-k}$ for every k, by automatic differentiation, and compare it with the gradient carried by the direct path alone, which multiplies the gradient at $\mathbf{c}_T$ elementwise by the product of the forget-gate values over the last k steps. Where do the two curves part, and which is larger?
7. code **No forget gate.** ▣ The original LSTM of Hochreiter and Schmidhuber (1997) had no forget gate; Gers et al. (2000) added it.
   1. Remove the forget gate from `LSTMScratch` by fixing $\mathbf{f}_t = \mathbf{1}$. Train language models with this layer and with the full layer as mixers of `SeqLM`, with `fit_lm` but 500 steps (the scratch loop is slow), and compare their perplexities.
   2. For both models, record the fraction of cell entries $c_{t,i}$ with $|\tanh(c_{t,i})| > 0.99$ at the last position of the validation windows. Use the result to explain the difference.
   3. Each training window starts from a zero state, so neither model runs more than 128 steps without a reset. Evaluate both models on 4,096 consecutive validation characters, once with the state carried across all of them and once with the state reset every 128 characters. Does the carried state hurt the model without a forget gate? Why?
8. code **Bidirectional leak.** ▣ A bidirectional network trained to predict the next character reads its target, as the end of Section 12.1.4.2 explains.
   1. Build a bidirectional mixer by running a second LSTM over the reversed sequence, reversing its outputs, concatenating them with those of a forward LSTM, and projecting the result back to width d. Train `SeqLM` with two such mixers using `fit_lm`, and compare its training and validation loss with those of the model with forward LSTMs.
   2. A leak-free bidirectional predictor of the character x_t combines the forward state after step $t-1$ with the backward state after step $t+1$, so it models the probability of x_t given all other characters of the window. Show with a sequence of two

binary tokens that the product of these conditionals over t need not sum to one over all sequences. What does this imply for using the product to score whole sequences, or to report a perplexity?

12.2 Linear Recurrences and State Space Models

The gated networks of Section 12.1 can keep information for hundreds of steps, but they still read a sequence one token after another, and each update needs the result of the one before. That order is natural when a model generates text, but during training the whole sequence is known in advance. Take a sequence of 4,096 tokens. A gated network performs 4,096 updates in a row. A GPU can carry out tens of thousands of multiply-adds at the same time, but it cannot start step 2,000 until step 1,999 has finished, so most of it waits. A transformer layer has no such chain: it processes all 4,096 positions in a few large matrix products. This difference in training speed, more than any difference in what the two kinds of model can represent, was a main reason why transformers replaced recurrent networks for large-scale language modeling. Can a recurrent network be trained on all positions at once?

The chain has a single cause. The gates of Section 12.1 read the previous state, and the state passes through a nonlinearity, so step t cannot be computed until step $t-1$ is known. If the gates read only the current input, the update becomes linear in the state. Linear updates can be grouped like the terms of a sum, and the states at all 4,096 positions can then be computed in $\log_2 4096 = 12$ rounds of batched arithmetic. When the model generates text, it still updates its state once per token, in constant memory, like any recurrent network.

A linear recurrence still has to decide how fast to forget, and a gate learned from scratch starts with a memory of about one step. The second half of the section borrows an answer from physics: write the recurrence as a differential equation that is sampled at discrete steps. Such a recurrence is stable for every value of its parameters, and it starts with memories that range from a step to a thousand steps. When its dynamics are the same at every position, it can also be computed as a convolution. These are the *state space models*. The next section makes their dynamics depend on the input.

12.2.1 Linear Recurrences

12.2.1.1 A Single Decaying Number

To see what a linear update gives us, we start with the smallest one. Consider a state h_t, a single number, that keeps a fraction a of its value at every step and adds the new input x_t:

$$h_t = a\, h_{t-1} + x_t, \qquad h_0 = 0. \tag{12.10}$$

This is the gated memory of Section 12.1 with both gates held fixed. With $a = 0.9$ and a single input $x_1 = 1$ followed by zeros, the state reads $1, 0.9, 0.81, 0.729, \ldots$: an input from k steps ago survives with weight a^k. Substituting the recurrence into itself shows that this holds for any input sequence,

$$h_t = \sum_{s=1}^{t} a^{t-s} x_s. \tag{12.11}$$

The state is a weighted sum of all past inputs, with weights that shrink geometrically with age. Two consequences follow. First, the sum does not mention h_{t-1}: every h_t is determined by the inputs alone, so in principle all of them can be computed at once. Second, a sets the length of the memory, and it sets the length of learning too. The weight of an input halves every $\ln 2/\ln(1/a)$ steps, about 7 steps for $a = 0.9$ and 693 for $a = 0.999$, and the gradient of h_t with respect to x_s is the same weight a^{t-s}. The decay that forgets an input also weakens the signal that would teach the model to use it. This is the vanishing gradient of Section 12.1, a product of $t - s$ Jacobians, except that the product is now a power of one number that we can read off directly.

12.2.1.2 Input-Dependent Decay

A memory of one number is too small for language. With a state $\mathbf{h}_t \in \mathbb{R}^d$ we can give each coordinate its own decay, collected in a vector $\mathbf{a} \in (0,1)^d$ that multiplies the state elementwise ($\odot$):

$$\mathbf{h}_t = \mathbf{a} \odot \mathbf{h}_{t-1} + \mathbf{b}_t.$$

Nothing new happens here. The coordinates do not interact, so this is d copies of Equation 12.10 side by side, each with its own half-life. We have also renamed the input term $\mathbf{b}_t$, because it will soon be a learned function of the input.

A fixed decay forgets at the same rate whatever the input says. The gates of Section 12.1 exist to make the rate a decision: keep the state through a run of uninformative tokens, and overwrite it when something important arrives. We can keep that ability, and keep the update linear in the state, by letting the decay depend on the current input:

$$\mathbf{h}_t = \mathbf{a}_t \odot \mathbf{h}_{t-1} + \mathbf{b}_t, \qquad \mathbf{a}_t = \mathbf{a}(\mathbf{x}_t), \quad \mathbf{b}_t = \mathbf{b}(\mathbf{x}_t). \tag{12.12}$$

Unrolling as before, the power a^{t-s} becomes the product of the decays met along the way:

$$\mathbf{h}_t = \sum_{s=1}^{t} \Big(\prod_{r=s+1}^{t} \mathbf{a}_r \Big) \odot \mathbf{b}_s. \tag{12.13}$$

An input survives for as long as the inputs after it choose to keep it. Because every $\mathbf{a}_r$ lies in $(0,1)$, the products can shrink but never grow: no past input is ever amplified, however long the sequence. Because $\mathbf{a}_t$ and $\mathbf{b}_t$ depend only on $\mathbf{x}_t$, they can be computed for all positions in one batched matrix product before the recurrence starts. What remains is the recurrence itself, an update that is *affine* in $\mathbf{h}_{t-1}$: an elementwise multiplication followed by an addition.

12.2.1.3 The Minimal GRU

The GRU of Section 12.1.3 almost has this form. Its update gate $\mathbf{z}_t$, reset gate $\mathbf{r}_t$, and candidate state $\tilde{\mathbf{h}}_t$ are

$$\begin{aligned}\mathbf{z}_t &= \sigma(\mathbf{W}_{\text{xz}}\mathbf{x}_t + \mathbf{W}_{\text{hz}}\mathbf{h}_{t-1} + \mathbf{b}_{\text{z}}),\\ \mathbf{r}_t &= \sigma(\mathbf{W}_{\text{xr}}\mathbf{x}_t + \mathbf{W}_{\text{hr}}\mathbf{h}_{t-1} + \mathbf{b}_{\text{r}}),\\ \tilde{\mathbf{h}}_t &= \tanh\left(\mathbf{W}_{\text{xh}}\mathbf{x}_t + \mathbf{W}_{\text{hh}}(\mathbf{r}_t \odot \mathbf{h}_{t-1}) + \mathbf{b}_{\text{h}}\right),\end{aligned}$$

and the new state is a blend of the old state and the candidate,

$$\mathbf{h}_t = \mathbf{z}_t \odot \mathbf{h}_{t-1} + (1 - \mathbf{z}_t) \odot \tilde{\mathbf{h}}_t.$$

The blend is already affine in $\mathbf{h}_{t-1}$, with decay $\mathbf{z}_t$. What stands in the way are the three places where $\mathbf{h}_{t-1}$ enters a gate or the candidate. Feng et al. (2024) removed them: the update gate and the candidate read only the input, and the reset gate, which no longer has a previous state to filter, disappears. The result is the *minimal GRU*, or *minGRU* (we keep a tanh on the candidate, which Feng et al. (2024) drop; an exercise compares the two):

$$\mathbf{z}_t = \sigma(\mathbf{W}_{\text{z}}\mathbf{x}_t + \mathbf{b}_{\text{z}}), \qquad \tilde{\mathbf{h}}_t = \tanh(\mathbf{W}_{\text{h}}\mathbf{x}_t + \mathbf{b}_{\text{h}}), \qquad \mathbf{h}_t = \mathbf{z}_t \odot \mathbf{h}_{t-1} + (1 - \mathbf{z}_t) \odot \tilde{\mathbf{h}}_t. \tag{12.14}$$

This is Equation 12.12 with decay $\mathbf{a}_t = \mathbf{z}_t$ and write $\mathbf{b}_t = (1 - \mathbf{z}_t) \odot \tilde{\mathbf{h}}_t$. The update gate is now the decay of a linear recurrence: a gate value of 0.9 keeps 90% of the state and admits 10% of the candidate. The layer is also smaller. With input and hidden width d, a GRU layer has $6d^2$ weights, three input matrices and three state matrices, and the minGRU has $2d^2$.

The minGRU gives something up. A GRU can decide to keep its state because of what the state contains; the minGRU decides only from the current token, and between tokens its state can only be scaled and added to. Stacking layers gives some of this back: the outputs of one layer, passed through a nonlinearity, are the inputs of the next, and nonlinearities between layers do not block parallel evaluation.

12.2.2 Parallel Scans

12.2.2.1 Composing Affine Steps

Computing $\mathbf{h}_1, \dots, \mathbf{h}_T$ from the coefficients $(\mathbf{a}_t, \mathbf{b}_t)$ is a relative of a familiar problem: compute all running totals $s_t = x_1 + \cdots + x_t$ of a sequence, its *prefix sums*. Prefix sums look sequential, yet they can be computed in parallel, because addition is associative: the total $(x_1 + x_2) + (x_3 + x_4)$ can be split into two halves that are computed independently and combined at the end. Any associative operation allows the same trick, and composing the steps of an affine recurrence is one.

Represent step t by its coefficient pair $(\mathbf{a}_t, \mathbf{b}_t)$, which stands for the map $\mathbf{h} \mapsto \mathbf{a}_t \odot \mathbf{h} + \mathbf{b}_t$. Applying step 1 and then step 2 gives $\mathbf{a}_2 \odot (\mathbf{a}_1 \odot \mathbf{h} + \mathbf{b}_1) + \mathbf{b}_2$, which is again a map of the same form. We write the composition as

$$(\mathbf{a}_1, \mathbf{b}_1) \bullet (\mathbf{a}_2, \mathbf{b}_2) = (\mathbf{a}_1 \odot \mathbf{a}_2,\ \mathbf{a}_2 \odot \mathbf{b}_1 + \mathbf{b}_2). \tag{12.15}$$

Read from left to right, $\bullet$ means "this step, then that one". It is associative because composing functions is associative, and the composite of any run of steps is again a single pair. This matters as much as associativity: a block of a thousand steps costs no more to store or

to combine than a single step. As an example, two steps with $a_1 = a_2 = 0.5$, $b_1 = 1$, and $b_2 = 2$ combine into $(0.25,\ 0.5 \cdot 1 + 2) = (0.25, 2.5)$. Running them one after the other from $h_0 = 0$ gives $h_1 = 1$ and $h_2 = 0.5 \cdot 1 + 2 = 2.5$, as the composite says.

Composing the first t pairs gives the map that carries $\mathbf{h}_0$ to $\mathbf{h}_t$. With $\mathbf{h}_0 = \mathbf{0}$ the state $\mathbf{h}_t$ is the $\mathbf{b}$-component of that composite, so all states together are the prefix "sums" of the pairs under $\bullet$. Computing all prefixes of a sequence under an associative operation is called a *scan*.

12.2.2.2 The Doubling Schedule

Figure 12.3 shows a parallel schedule for eight steps. In the first round, every position combines with its predecessor, so it holds the composite of two steps. In the second round, every position combines with the position two back and then holds four steps; in the third, with the position four back, and then holds eight. After $\lceil \log_2 T \rceil$ rounds every position holds its full prefix. Each round is one elementwise multiply-add over the whole sequence, the kind of work a GPU does well.

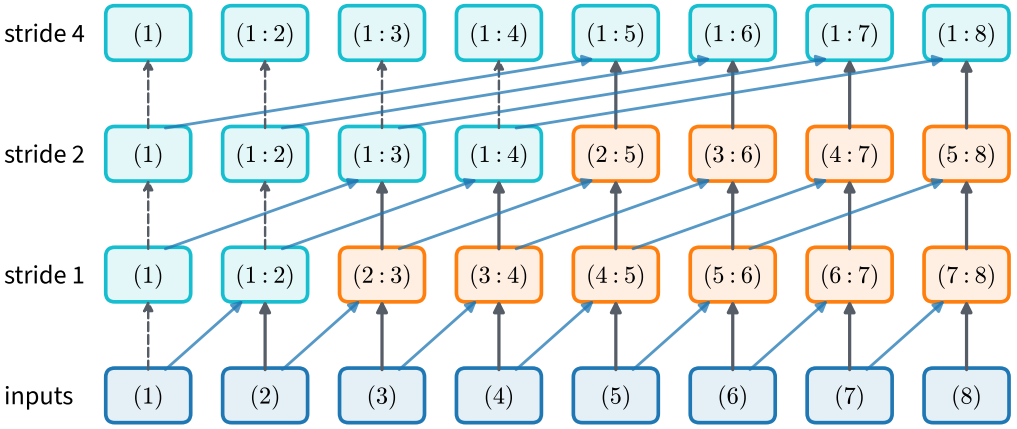


Figure 12.3: A parallel scan on eight steps. Each box shows which steps its value combines, $(1{:}4)$ meaning steps 1 to 4. In every round each position combines with the position a fixed stride back (blue arrows), and the stride doubles from round to round; dashed arrows copy a value unchanged. Teal boxes hold complete prefixes. After $\log_2 8 = 3$ rounds every position does.

This schedule, due to Hillis and Steele (1986), performs about $T \log_2 T$ combines, a factor $\log_2 T$ more than the $T - 1$ steps of a loop, in exchange for $\log_2 T$ rounds instead of $T - 1$. A schedule of Blelloch (1990) needs only about $2T$ combines in about $2 \log_2 T$ rounds. The difference in total work matters once a sequence is long enough to keep the whole GPU busy.

12.2.2.3 Implementation

`linear_scan` implements the doubling schedule. The writes $\mathbf{b}_t$ have shape (batch size, number of steps, ...), with time along the second axis. The decays are broadcast to the same shape, so decays that are the same at every step can be passed once, without the batch and time axes. An optional initial state is folded into the first step, since $\mathbf{h}_1 = \mathbf{a}_1 \odot \mathbf{h}_0 + \mathbf{b}_1$.

PyTorch has no scan primitive in its stable interface, so we write one, `associative_scan`, with the interface of JAX's: it takes the combine as a function and a tuple of tensors with time along the second axis. Each round of the doubling schedule combines the sequence with a copy of itself shifted by the stride, a few tensor operations per round. `linear_scan` passes it the combine Equation 12.15; the delta rule of Section 12.5 will pass it another.

```
def associative_scan(combine, elems):
    """All prefixes of elems under an associative combine; time is axis 1."""
    s = 1
    while s < elems[0].shape[1]:  # Combine every position with the one s back
        earlier = tuple(x[:, :-s] for x in elems)
        later = tuple(x[:, s:] for x in elems)
        elems = tuple(torch.cat([x[:, :s], y], 1)
                      for x, y in zip(elems, combine(earlier, later)))
        s *= 2
    return elems

def linear_scan(a, b, h0=None):
    """All states of h_t = a_t * h_{t-1} + b_t; time is the second axis."""
    a = a.expand_as(b)  # Decays may omit axes on which they do not vary
    if h0 is not None:  # Fold the initial state into the first step
        b = torch.cat([a[:, :1] * h0.unsqueeze(1) + b[:, :1], b[:, 1:]], 1)
    def combine(earlier, later):
        (a1, b1), (a2, b2) = earlier, later
        return a1 * a2, a2 * b1 + b2
    return associative_scan(combine, (a, b))[1]
```

We check the scan against a loop that applies the recurrence one step at a time, on random decays in $(0, 1)$ and a length that is not a power of two.

```
def sequential_scan(a, b):
    h, states = torch.zeros_like(b[:, 0]), []
    for a_t, b_t in zip(a.unbind(1), b.unbind(1)):
        h = a_t * h + b_t
        states.append(h)
    return torch.stack(states, 1)

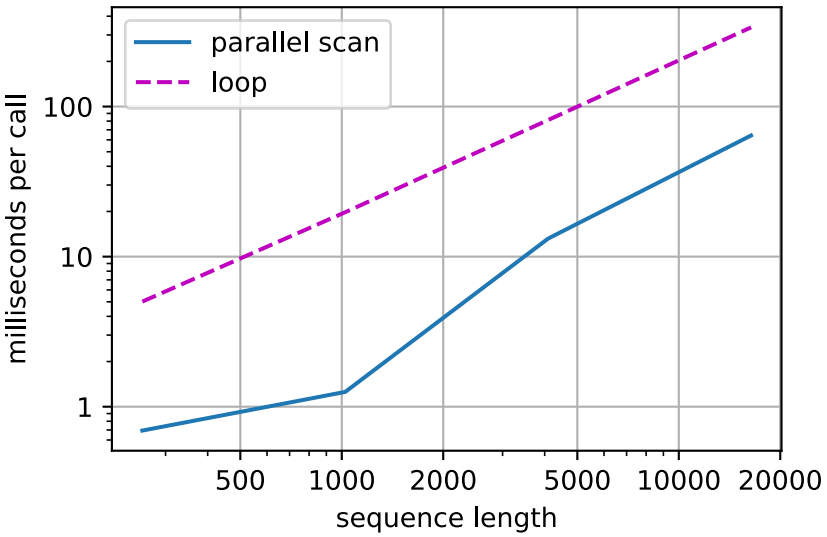
a, b = torch.rand(4, 100, 8), torch.randn(4, 100, 8)
print(f'max deviation: '
      f'{(linear_scan(a, b) - sequential_scan(a, b)).abs().max():.1e}')
```

```
max deviation: 4.8e-07
```

The two agree to the rounding error of 32-bit floating point. The scan groups the multiplications and additions differently from the loop, so the results differ in the last digits; they are the same computation.

How much faster is the scan? We time both on a GPU for sequences of 256 to 16,384 steps, with a batch of 32 sequences of 128 coordinates each. `d2l.Benchmark`, which Section 13.1 introduces, calls a function a few times after a warm-up call and reports the average time, waiting for the GPU to finish before it stops the clock.

```
device = d2l.try_gpu()
lengths, times = [256, 1024, 4096, 16384], [[], []]
for T in lengths:
    a = torch.rand(32, T, 128, device=device)
    b = torch.randn(32, T, 128, device=device)
    for i, scan in enumerate([linear_scan, sequential_scan]):
        bench = d2l.Benchmark(lambda: scan(a, b), warmup=1, repeats=3)
        times[i].append(1e3 * bench.time)
d2l.plot(lengths, times, 'sequence length', 'milliseconds per call',
         legend=['parallel scan', 'loop'], xscale='log', yscale='log',
         figsize=(4.5, 3))
```



The loop's time grows in proportion to the length: each step is a separate small computation that must finish before the next one starts. The scan is faster at every length, by a factor between a few and about a hundred, depending on the length. Its time does not follow the length: while its rounds of arithmetic are too small to keep the GPU busy, more positions cost little extra time, and once the GPU is full, the total amount of work sets the time. Where this happens depends on how the scan is compiled. The scan changes how we train, not how we generate: at inference the model still applies Equation 12.12 once per token, in constant memory.

12.2.2.4 A Linear Recurrent Language Model

The minGRU Equation 12.14 is a few lines around `linear_scan`. It maps a minibatch of sequences to the states at every step, so it can serve as the sequence mixer of `d2l.SeqLM`, the residual language model of Section 12.1: an embedding, a stack of blocks that each apply a mixer and a position-wise MLP, and a prediction head.

```
class MinGRU(nn.Module):
    """The minimal GRU: its gate and candidate read only the input."""
    def __init__(self, num_inputs, num_hiddens):
```

```
        super().__init__()
        self.W_z = nn.Linear(num_inputs, num_hiddens)  # Update gate (decay)
        self.W_h = nn.Linear(num_inputs, num_hiddens)  # Candidate state
        d2l.init_lecun(self)

    def coefficients(self, X):
        """Decays a_t and writes b_t of the linear recurrence."""
        Z = torch.sigmoid(self.W_z(X))
        return Z, (1 - Z) * torch.tanh(self.W_h(X))

    def forward(self, X):  # X: (batch_size, num_steps, num_inputs)
        return linear_scan(*self.coefficients(X))
```

We train it with the recipe of Section 12.1: character-level Penn Treebank in windows of 128 characters, two blocks of width 128, and 3,000 steps of AdamW, all wrapped in `d2l.fit_lm`, which returns the validation perplexity. With the same recipe, the GRU reached a perplexity of about 3.0 and the plain recurrent network about 3.1.

```
data = d2l.PennTreebank(batch_size=64, num_steps=128)
```

```
model = d2l.SeqLM(len(data.vocab), 128, [MinGRU(128, 128) for _ in range(2)])
print(f'minGRU perplexity: {d2l.fit_lm(model, data):.2f}')
```

```
minGRU perplexity: 3.19
```

The minGRU reaches a perplexity of about 3.2, a little behind the GRU's 3.0, with a third of the GRU's weights per layer. On this task, taking the state out of the gates costs a few percent.

Because the minGRU's decay is its update gate, we can look inside the trained model. Two questions are easy to answer. First, does the model use its memory at all? We compare its perplexity with that of the same model whose layers reset their state to zero before every step, so that $\mathbf{h}_t = (1-\mathbf{z}_t)\odot\tilde{\mathbf{h}}_t$ and each prediction sees only the current character, as transformed by the layers. Second, how often does a gate hold the state rather than overwrite it?

```
class StateReset(nn.Module):
    """A minGRU layer whose state is reset to zero before every step."""
    def __init__(self, layer):
        super().__init__()
        self.layer = layer

    def forward(self, X):
        return self.layer.coefficients(X)[1]

layers = model.mixers
model.mixers = nn.ModuleList([StateReset(layer) for layer in layers])
ppl = d2l.lm_perplexity(model, data.val_dataloader())
print(f'perplexity with the state reset: {ppl:.2f}')
model.mixers = layers
```

```
perplexity with the state reset: 10.81
```

Resetting the state raises the perplexity from about 3.2 to more than 10, so the model relies heavily on what it remembers: the next character is hard to predict from the current one alone.

To read the gates, we need the input of the first layer: the token embeddings, normalized as the first block normalizes them. We give `SeqLM` a method for it, since the next section reads it again.

```
@d2l.add_to_class(d2l.SeqLM)
def first_mixer_input(self, X):
    """What the first mixer reads: the normalized token embeddings."""
    return self.norms[0](self.embedding(X))
```

```
X, _ = next(iter(data.val_dataloader()))
X = X.to(next(model.parameters()).device)
with torch.inference_mode():  # Gates of the first layer on validation text
    Z = layers[0].coefficients(model.first_mixer_input(X))[0]
print(f'gates above 0.5: {(Z > 0.5).float().mean():.0%}, '
      f'above 0.9: {(Z > 0.9).float().mean():.0%}')
```

```
gates above 0.5: 53%, above 0.9: 25%
```

About half of the gate values in the first layer exceed 0.5, and about a quarter exceed 0.9, which keeps the state with a half-life of 7 steps or more. Each gate value is computed from the current character, so how long an input survives depends on the characters that follow it, as Equation 12.13 describes; an exercise asks how much the gates of a unit vary from character to character.

The minGRU shows that a gated recurrence can be linear in its state, and therefore trained with a scan, at a small cost in perplexity. It leaves two questions open. Its gates start near $\sigma(0) = 1/2$, a half-life of one step, so an input 20 steps back initially reaches the state, and sends its gradient back, with weight 2^{-20}, about one in a million; nothing in the model offers longer memories from the start. And its memory holds only one number per hidden unit. The rest of the section derives the decay from a continuous-time model, which gives stable decays with a spread of timescales from the first step of training, and a larger state at little extra cost.

12.2.3 State Space Models

The minGRU trains in parallel, but like the gated networks of Section 12.1 it must learn from scratch how long to remember. A linear recurrence holds a value for a thousand steps only if its decay stays within about 0.001 of one for all thousand steps: a decay of 0.999 has a half-life of 693 steps. The minGRU starts with decays near one half and must find such values by gradient descent, from gradients that, at the start, have been halved a thousand times. Nothing in the recurrence says which memory lengths a model will need, either. The subject of a sentence matters for a few words, the topic of a paragraph for hundreds. Physics and control theory have long described systems whose state changes continuously in time, such as a cooling cup of coffee or the charge on a capacitor. Reading such a system at discrete steps gives a linear recurrence whose decays are stable for every value of the parameters and can be initialized to cover many timescales at once.

12.2.3.1 Continuous-Time Dynamics

The smallest case is one number $h(t)$ that changes at a rate proportional to its own value, driven by an input $x(t)$:

$$h'(t) = \lambda\, h(t) + x(t). \tag{12.16}$$

Without input, the solution is $h(t) = e^{\lambda t} h(0)$. For $\lambda < 0$ the state decays to zero, faster for larger $|\lambda|$; with $\lambda = -1$, a fraction $e^{-1} \approx 0.37$ remains after one unit of time. For $\lambda > 0$ it grows without bound. The parameter λ is a rate: it says how fast the state forgets per unit of time, not per step.

A *state space model* (SSM) generalizes this to a vector state $\mathbf{h}(t) \in \mathbb{R}^N$ that is driven by a scalar input $x(t)$ and read out as a scalar output $y(t)$:

$$\mathbf{h}'(t) = \mathbf{A}\,\mathbf{h}(t) + \mathbf{B}\,x(t), \qquad y(t) = \mathbf{C}\,\mathbf{h}(t) + D\,x(t). \tag{12.17}$$

The *state matrix* $\mathbf{A} \in \mathbb{R}^{N \times N}$ sets the dynamics, $\mathbf{B} \in \mathbb{R}^N$ says how the input enters the state, $\mathbf{C} \in \mathbb{R}^{1 \times N}$ reads the state, and D passes the input straight through. Control engineers use such models for motors and circuits; the name says that the system is described by the space of its states. We take $\mathbf{A} = \operatorname{diag}(\lambda_1, \dots, \lambda_N)$ diagonal, so that the model is N independent copies of Equation 12.16, each with its own rate, read out together by $\mathbf{C}$. A sequence of d-dimensional vectors gets one such SSM per coordinate, or *channel*, so the state of a layer holds $d \times N$ numbers, N times as many as a minGRU of the same width.

12.2.3.2 Discretization

To process a sequence, we read the continuous system at steps of length Δ. If the input stays constant during each step, a rule called the *zero-order hold*, then Equation 12.16 can be solved exactly over one step, and the result is a linear recurrence:

$$h_t = \bar{a}\, h_{t-1} + \bar{b}\, x_t, \qquad \bar{a} = e^{\Delta\lambda}, \qquad \bar{b} = \frac{e^{\Delta\lambda} - 1}{\lambda}. \tag{12.18}$$

The state decays by $e^{\Delta\lambda}$ over the step, and the input, which enters continuously during the step, contributes $\int_0^{\Delta} e^{\lambda\tau}\, d\tau = \bar{b}$ times its value. For $\lambda = -1$ the two coefficients are $\bar{a}$ and $1 - \bar{a}$, so the update reads $h_t = \bar{a}\, h_{t-1} + (1 - \bar{a})\, x_t$: the blend of the GRU, with $\bar{a}$ in the role of the update gate. The step size decides the gate. With $\Delta = 0.01$ the state keeps $\bar{a} = 0.990$ of its value, a half-life of 69 steps, and admits 1% of the input; with $\Delta = 0.1$ it keeps 0.905, a half-life of 7 steps; with $\Delta = 1$ it keeps 0.37 and is mostly overwritten. A small step size means a long memory.

The same formula guarantees stability. For every $\lambda < 0$ and every $\Delta > 0$, the decay $\bar{a} = e^{\Delta\lambda}$ lies in $(0, 1)$. If we store the rate as $\lambda = -e^{\theta}$ and learn θ, then no value of the parameters, and no gradient step, can produce a recurrence that grows. Figure 12.4 shows the map from rates to decays. Rates may also be complex, $\lambda = -\alpha + i\omega$, in which case the state decays while it rotates, a damped oscillation; the models of this chapter use real rates.

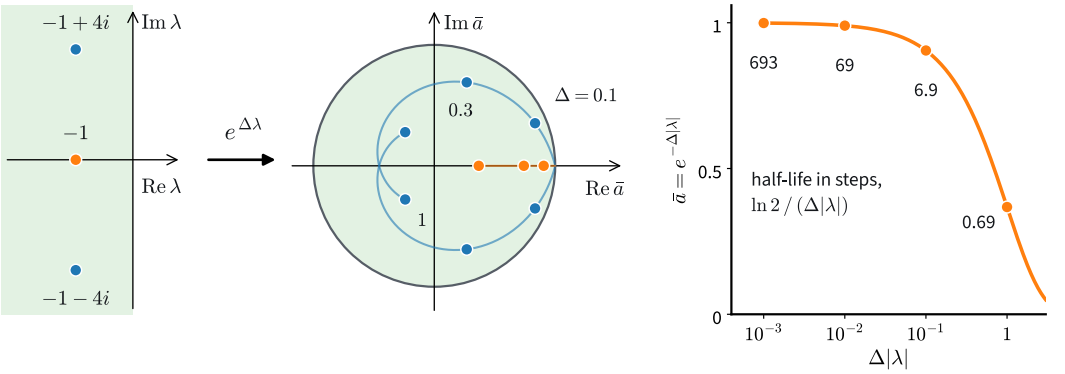


Figure 12.4: Zero-order hold maps a rate λ to a decay per step $\bar{a} = e^{\Delta\lambda}$. Left and middle: every rate with negative real part (shaded) lands inside the unit circle, for every step size. The curves trace $\lambda = -1 \pm 4i$ for Δ from 0 to 1, with dots at $\Delta = 0.1, 0.3, 1$; the orange dots show $\lambda = -1$ at the same step sizes. Right: for real rates, the decay as a function of $\Delta|\lambda|$, labeled with the half-life in steps.

With a diagonal $\mathbf{A}$, the zero-order hold applies to each state separately: $\bar{a}_n = e^{\Delta\lambda_n}$ and $\bar{b}_n = (e^{\Delta\lambda_n} - 1)b_n/\lambda_n$. (For small $\Delta\lambda_n$, the difference $e^{\Delta\lambda_n} - 1$ loses digits in floating point, so the code computes it with `expm1`, which evaluates $e^x - 1$ accurately.)

Discretization also tells us how to initialize. State n forgets with a time constant of $1/(\Delta|\lambda_n|)$ steps. The S4D model of Gu, Gupta, et al. (2022), in the variant with real rates that we build below, sets $\lambda_n = -(n+1)$ for $n = 0, \ldots, N-1$ and draws one step size per channel, log-uniformly between 0.001 and 0.1. With $N = 16$ states, the time constants then range from under one step ($\Delta = 0.1$, $\lambda = -16$) to 1,000 steps ($\Delta = 0.001$, $\lambda = -1$): before training, the layer already holds memories of every length in between. These rates come from the HiPPO theory of Gu et al. (2020), which designs $\mathbf{A}$ so that the state stores the coefficients of a polynomial approximation of the input's history; the diagonal rates above approximate that design.

12.2.3.3 The Convolutional Form

The scan is one way to compute all states at once, and dynamics that do not change along the sequence allow a second. When $\bar{a}_n$, $\bar{b}_n$, and $\mathbf{C}$ are the same at every step, the model is *linear time-invariant*, and unrolling it gives the outputs directly in terms of the inputs. State n holds $\sum_{k \geq 0} \bar{a}_n^k \bar{b}_n x_{t-k}$, as in Equation 12.11, and the readout adds the states with weights c_n:

$$y_t = \sum_{k=0}^{t-1} K_k \, x_{t-k} + D x_t, \qquad K_k = \sum_{n=1}^{N} c_n \bar{a}_n^k \bar{b}_n. \tag{12.19}$$

The output is a *causal convolution* of the input with the kernel $\mathbf{K} = (K_0, K_1, \ldots)$: every input contributes with a weight that depends only on how many steps ago it arrived. The kernel is a sum of N decaying exponentials, one per state. A single state with $\lambda = -1$, $\Delta = 0.1$, and $c = 1$ gives $K_k = 0.095 \cdot 0.905^k$, the weights of an exponential moving average.

One model therefore has three equivalent forms (Figure 12.5). The differential equation is where we choose the parameters and see that they are stable. The recurrence is how the model generates, one step at a time in constant memory. The convolution is a second way to train in parallel: the convolution of two sequences of length T can be computed with the fast Fourier transform (FFT) in $O(T \log T)$ operations, by transforming both, multiplying them, and transforming back. The S4 model of Gu, Goel, et al. (2022) trained this way on sequences of tens of thousands of steps, and it solved benchmarks whose dependencies span thousands of steps. The convolution requires dynamics that are the same at every step; the scan does not.

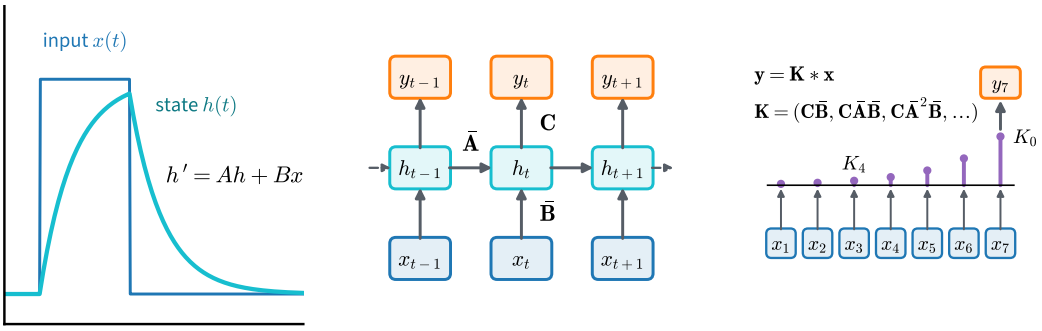


Figure 12.5: Three forms of one state space model. Left: the continuous-time state responds to an input pulse. Middle: the discretized system is a linear recurrence, one step per token. Right: unrolled, the same system is a causal convolution whose kernel is the response to a single pulse.

12.2.4 S4D

12.2.4.1 The S4D Layer

The `S4D` layer runs one diagonal SSM with N states per channel. It stores the rates as $\lambda_n = -e^{\theta_n}$ and one step size per channel as $\log \Delta$, initialized as described above. Following Gu, Gupta, et al. (2022), it fixes $\mathbf{B} = \mathbf{1}$ and lets the learned readout $\mathbf{C}$ absorb its scale. We start with the parameters and the method `discretize`, which applies the zero-order hold Equation 12.18 to every state of every channel.

```
class S4D(nn.Module):
    """A diagonal state space layer: N states per channel, fixed dynamics."""
    def __init__(self, num_hiddens, num_states=16, dt_min=1e-3, dt_max=1e-1):
        super().__init__()
        d, N = num_hiddens, num_states
        # Rates lambda_n = -(n + 1), stored as log(-lambda) to stay negative
        self.log_neg_rate = nn.Parameter(
            torch.log(torch.arange(1., N + 1)).repeat(d, 1))
        self.log_dt = nn.Parameter(  # Log-uniform in [dt_min, dt_max]
            torch.rand(d, 1) * math.log(dt_max / dt_min) + math.log(dt_min))
        self.C = nn.Parameter(torch.randn(d, N) / math.sqrt(N))
        self.D = nn.Parameter(torch.ones(d))

    def discretize(self):
```

```
        """Zero-order hold with B = 1: decays a_bar and input weights b_bar."""
        rate = -torch.exp(self.log_neg_rate)  # (d, N)
        dt_rate = torch.exp(self.log_dt) * rate
        return torch.exp(dt_rate), torch.expm1(dt_rate) / rate
```

For training, `forward` computes the kernel Equation 12.19 of every channel from the discretized parameters and convolves it with the input by the FFT. Both sequences are padded to twice their length, because the FFT computes a *circular* convolution, in which the end of a sequence would otherwise wrap around and leak into its beginning.

```
@d2l.add_to_class(S4D)
def forward(self, X):  # X: (batch_size, num_steps, d)
    T = X.shape[1]
    a_bar, b_bar = self.discretize()
    powers = a_bar.unsqueeze(-1) ** torch.arange(T, device=X.device)
    K = ((self.C * b_bar).unsqueeze(-1) * powers).sum(1)  # Kernel, (d, T)
    Y = torch.fft.irfft(torch.fft.rfft(X, n=2 * T, dim=1) *
                        torch.fft.rfft(K.T, n=2 * T, dim=0), n=2 * T, dim=1)
    return Y[:, :T] + self.D * X
```

For generation, `step` advances the recurrence Equation 12.18 of every state by one token and reads the output. Its state has $d \times N$ numbers per sequence, whatever the length of the text generated so far.

```
@d2l.add_to_class(S4D)
def step(self, x, h):
    """One step: x is (batch_size, d); the state h is (batch_size, d, N)."""
    a_bar, b_bar = self.discretize()
    h = a_bar * h + b_bar * x.unsqueeze(-1)
    return (h * self.C).sum(-1) + self.D * x, h
```

We check that the three forms agree on a random layer: the FFT convolution of `forward`, the parallel scan of the recurrence, and the recurrence run one step at a time with `step`.

```
layer = S4D(8, num_states=4)
X = torch.randn(2, 50, 8)  # (batch_size, num_steps, d)
with torch.no_grad():
    a_bar, b_bar = layer.discretize()
    H = linear_scan(a_bar, b_bar * X.unsqueeze(-1))  # (batch, T, d, N)
    Y_scan = (H * layer.C).sum(-1) + layer.D * X
    h, Y_step = torch.zeros(2, 8, 4), []
    for x in X.unbind(1):
        y, h = layer.step(x, h)
        Y_step.append(y)
    Y_conv = layer(X)
print(f'convolution vs scan: {(Y_conv - Y_scan).abs().max():.1e}, '
      f'convolution vs steps: {(Y_conv - torch.stack(Y_step, 1)).abs().max():.1e}')
```

```
convolution vs scan: 1.2e-07, convolution vs steps: 1.2e-07
```

The three forms agree to rounding error. The FFT and the scan both train in parallel; for

the time-invariant S4D, the FFT does so without storing the $d \times N$ state at every position, only the kernel of length T per channel.

12.2.4.2 Long-Range Recall

Do the timescales of the initialization matter? We return to the delayed recall task of Section 12.1: a sequence starts with one of eight symbols, and after $T-1$ filler tokens the model must name it. There, no network with gates that start near one half kept the symbol for 50 steps. Here the fillers are all the same token, and the delays are 100 and 1,000 steps. Identical fillers isolate the question of timescales: a model with fixed dynamics cannot ignore random fillers, which the next section addresses, but identical ones add only a constant to its state. We compare an LSTM and a minGRU whose gates start near one half, one S4D layer with 64 channels of 16 states, and two controls. The first is an S4D layer whose step sizes start between 0.1 and 1, so that its time constants are at most about 10 steps. The second is a minGRU whose gate biases are set so that its decays start at the values $e^{-\Delta}$ of the slowest state of S4D, with Δ log-uniform between 0.001 and 0.1.

```
def init_timescales(layer, dt_min=1e-3, dt_max=1e-1):
    """Start a minGRU's gates at the decays exp(-dt) of S4D."""
    dt = dt_min * (dt_max / dt_min) ** torch.rand(layer.W_z.bias.shape)
    with torch.no_grad():  # sigmoid(b) = exp(-dt)
        layer.W_z.bias.copy_(-torch.log(torch.expm1(dt)))
    return layer
```

We reuse the model and the training function `d2l.train_delayed_recall` of Section 12.1: an embedding, the sequence layer, and a classifier read at the last position, trained with Adam on fresh batches. Here the learning rate is 0.01.

```
# (layer, embedding width): S4D keeps its width; the others map 32 to 64
layers = {
    'LSTM': lambda: (d2l.Recurrent(d2l.init_lecun(
        nn.LSTM(32, 64, batch_first=True))), 32),
    'minGRU': lambda: (MinGRU(32, 64), 32),
    'S4D': lambda: (S4D(64), 64),
    'S4D, short timescales': lambda: (S4D(64, dt_min=0.1, dt_max=1.0), 64),
    'minGRU, S4D timescales': lambda: (init_timescales(MinGRU(32, 64)), 32)}
print(' ' * 23, '  T=100  T=1000')
for name, layer in layers.items():
    accuracy = []
    for T in [100, 1000]:
        layer_T, num_inputs = layer()
        model = d2l.RecallModel(layer_T, 64, num_inputs=num_inputs)
        accuracy.append(d2l.train_delayed_recall(model, T, blank_fillers=True,
                                                 lr=1e-2))
    print(f'{name:23s}', ' '.join(f'{a:6.2f}' for a in accuracy))
```

```
                        T=100  T=1000
LSTM                    0.12   0.12
minGRU                  0.14   0.12
S4D                     1.00   1.00
S4D, short timescales   1.00   0.13
minGRU, S4D timescales  1.00   1.00
```

The LSTM and the minGRU, with gates that start near one half, are at chance already after 100 steps. S4D recalls the symbol after 1,000 steps. The two controls show where its advantage comes from. With short time constants, S4D recalls the symbol after 100 steps but not after 1,000; given the time constants of S4D, the minGRU recalls it after 1,000 steps too. On this task the initialization decides: memories of many lengths, present from the first step of training. The continuous-time view supplies them, together with decays that are stable for every value of the parameters, while the convolution makes training fast but is not needed for the result.

12.2.4.3 Language Modeling

An S4D layer is linear and mixes no information across channels. In `d2l.SeqLM` each S4D layer is followed by the position-wise MLP, which mixes the channels and supplies the nonlinearity. We train two such blocks with 16 states per channel, with the recipe of the minGRU above.

```
model = d2l.SeqLM(len(data.vocab), 128, [S4D(128) for _ in range(2)])
print(f'S4D perplexity: {d2l.fit_lm(model, data):.2f}')
```

```
S4D perplexity: 3.34
```

S4D reaches a perplexity of about 3.35, behind the minGRU's 3.2 and the GRU's 3.0, although each of its layers holds 16 times as many numbers in its state as a minGRU layer. Its dynamics are the same at every step, so an input's weight in the state depends only on how many steps ago it arrived. Whether the character four steps back ended a word or sat in its middle, its influence on the present is the same fixed number K_4. The minGRU, whose decays are chosen by each input, can hold its state through some characters and overwrite it at others; S4D cannot.

12.2.5 Summary

A recurrence whose update is linear in its state, $\mathbf{h}_t = \mathbf{a}_t \odot \mathbf{h}_{t-1} + \mathbf{b}_t$, keeps a weighted sum of its past inputs, with weights that are products of decays. Because affine steps compose into affine steps, a parallel scan computes all states in about $\log_2 T$ rounds of batched arithmetic. The minGRU obtains such a recurrence from the GRU by letting its gate and candidate read only the input; it comes within a few percent of the GRU's perplexity, and resetting its state shows that it relies on its memory. A state space model derives the decays from a continuous-time system read at steps of size Δ: the decays $e^{\Delta\lambda}$ are stable for every value of the parameters, and the initialization of S4D gives every layer memories from one step to a thousand. With fixed dynamics the model is also a convolution, which the FFT computes. On long-range recall, this initialization lets S4D, and a minGRU with the same timescales, keep a token for 1,000 steps where gates that start at one half fail. The weights of S4D, however, depend only on distance, so it cannot choose what to keep; the next section lets the dynamics depend on the input.

12.2.6 Exercises

1. **Closure under composition.** A parallel scan needs more than associativity: the composite of any run of steps must be described by as many numbers as a single step, so that every combine costs the same.

   1. Generalize Equation 12.15 to the recurrence $\mathbf{h}_t = \mathbf{A}_t\mathbf{h}_{t-1} + \mathbf{b}_t$ with dense matrices $\mathbf{A}_t \in \mathbb{R}^{d\times d}$, keeping the factors of each product in the right order. Count the multiply-adds of one combine and of one sequential step. Compare the total work of the doubling scan, about $T \log_2 T$ combines, with that of the loop at $T = 4096$ and $d = 64$. What does the comparison suggest about dense transition matrices?
   2. For the scalar step $h \mapsto \tanh(w_t h + c_t)$, show that the composite of two steps with $w_1, w_2 \neq 0$ is not of the form $h \mapsto \tanh(wh + c)$ for any w and c. What would a scan have to store to represent the composite of k steps?
2. code **Candidate nonlinearity.** `MinGRU` keeps a tanh on the candidate $\tilde{\mathbf{h}}_t$, whereas Feng et al. (2024) use a linear candidate.
   1. Starting from $\mathbf{h}_0 = \mathbf{0}$, write each coordinate of $\mathbf{h}_t$ as a weighted sum of the candidates $\tilde{\mathbf{h}}_1, \dots, \tilde{\mathbf{h}}_t$. Show that the weights are nonnegative and sum to at most one, and derive a bound on $|\mathbf{h}_t|$ that holds at every length, with and without the tanh.
   2. Remove the tanh, retrain with `d2l.fit_lm`, and compare the perplexity with that of `MinGRU`. Relate what you observe to your bound.
3. code **A minimal LSTM.** Apply the same two cuts to the LSTM of Section 12.1: compute its input, forget, and output gates and its input node from $\mathbf{x}_t$ alone, and keep its cell update $\mathbf{c}_t = \mathbf{f}_t \odot \mathbf{c}_{t-1} + \mathbf{i}_t \odot \tilde{\mathbf{c}}_t$.
   1. Show that only the cell needs a scan. Implement the layer as a mixer, train it with `d2l.fit_lm`, and compare its perplexity with that of `MinGRU`.
   2. Unlike the minGRU, this layer does not tie its input gate to its forget gate. Show that if $|\tilde{\mathbf{c}}_t| < 1$ and every forget gate stays below $f_{\max} < 1$, the cell stays below $1/(1 - f_{\max})$ in magnitude, and give gate values under which it approaches this bound. What bound holds if the gates are replaced by $\mathbf{f}_t/(\mathbf{f}_t + \mathbf{i}_t)$ and $\mathbf{i}_t/(\mathbf{f}_t + \mathbf{i}_t)$, as Feng et al. (2024) propose?
4. code **State passing.** `linear_scan` accepts an initial state, so a long sequence can be processed in pieces.
   1. Split a sequence of length T into chunks of length τ, run the scan on each chunk starting from the final state of the previous one, and check that the concatenated states match a single call on the whole sequence.
   2. Now stop gradients through the carried state at every chunk boundary. The states do not change. Which contributions to the gradient of the loss at step t disappear, and which dependencies can the trained model still use when it generates text? Compare with truncated backpropagation through time (Section 8.6).
5. code **How long must memories start?** `init_timescales` starts the minGRU's gates at half-lives of up to about 700 steps.
   1. What bias b gives a gate $\sigma(b)$ with a half-life of k steps?
   2. Initialize the gates with half-lives log-uniform between 1 and $k_{\max}$ steps and repeat the long-range recall experiment at 1,000 steps for $k_{\max} \in \{10, 30, 100, 300\}$. What is the smallest $k_{\max}$ that solves the task? Relate it to the size of the gradient that reaches the first token at initialization.
   3. Measure how much the gates of each unit vary across the characters of a validation window in the trained language model. Do some units hold their state on some characters and overwrite it on others?
6. code **Work-efficient scans.** ⊠ A recursive schedule does less work than the doubling schedule: combine adjacent pairs, which halves the sequence; scan the half-length sequence recursively, which yields every second prefix; then obtain each remaining prefix with one more combine.
   1. For $T = 2^k$, count the combines and the rounds of combines of each schedule,

and compare both with the $T-1$ steps of the loop at $T = 16384$.
   2. Implement the recursive schedule with slicing, check it against `sequential_scan`, and time it against `linear_scan` at the lengths used above. Does the ratio of times follow the ratio of combines? Explain why it might not at short lengths.
7. code **Discretization rules.** The zero-order hold is one of several ways to turn Equation 12.16 into a recurrence. The forward Euler rule uses $\bar{a} = 1 + \Delta\lambda$, the backward Euler rule $\bar{a} = 1/(1 - \Delta\lambda)$, and the bilinear rule $\bar{a} = (1 + \Delta\lambda/2)/(1 - \Delta\lambda/2)$.
   1. For each rule, find the step sizes Δ for which every $\lambda < 0$ gives $|\bar{a}| < 1$. Which rule can make a stable continuous system unstable?
   2. Show that backward Euler with $\lambda = -1$ gives the update $h_t = zh_{t-1} + (1-z)x_t$ with $z = \sigma(-\log \Delta)$, the blend of a GRU whose update gate has the negative logarithm of the step size as its pre-activation.
   3. Simulate the response of each rule to a unit step input for $\lambda = -1$ and $\Delta \in \{0.1, 1, 3\}$, and compare with the exact solution $1 - e^{-t}$.
8. code **Convolution by the FFT.** `S4D.forward` pads the sequence and the kernel to length $2T$ before the FFT.
   1. Remove the padding, so that the FFT has length T, and compare the outputs with those of `step`. Which outputs are wrong, and what has been added to them?
   2. What is the shortest padded length that gives the exact causal convolution? Why do implementations often round it up to a power of two?

12.3 Selective State Space Models

The S4D layer of Section 12.2 treats every input alike: an input from k steps ago reaches the output through the fixed number K_k of the convolution kernel, whatever that input was. Whether the token ten steps ago was the key fact of a paragraph or a comma, its influence on the present is the same. Language is not like this. Most tokens can be forgotten quickly, and a few, such as a name, a number, or the subject of a sentence, must be kept for a long time, at positions that differ from one sentence to the next. A memory that keeps everything for the same time fills up with commas; a memory that forgets quickly loses the name. Can a state space model decide, token by token, what to keep and for how long?

Two models of the previous section each have half of a solution. The minGRU chooses its decay from the current input, so it can keep one token and overwrite the next, but its gates start with a memory of about one step, and its state holds one number per channel. S4D starts with memories of a step to a thousand steps and holds N numbers per channel, but its dynamics are the same for every input. *Selective* state space models, introduced with the Mamba architecture by Gu and Dao (2023), combine the two: the step size Δ of the discretized SSM, and the vectors that write to and read from the state, become functions of the current input. A small step size keeps the state and ignores the token; a large one overwrites the state with it. With input-dependent dynamics, Mamba matched or exceeded transformers of the same size in language modeling. The convolution of S4D no longer applies, because the dynamics change from step to step, but the scan of Section 12.2 does. We test these models on a task that requires choosing what to keep, build the Mamba layer, measure where its state comes from, and train it as a language model. The state of a selective SSM turns out to be a matrix written one outer product at a time, which the next section studies in its own right.

12.3.1 The Selectivity Problem

12.3.1.1 Selective Copying

The *selective copying* task of Gu and Dao (2023) isolates the decision of what to keep (Figure 12.6). A sequence consists of 1,024 filler tokens among which four symbols, drawn from eight, sit at random positions. Four query tokens follow, and at each of them the model must output the next symbol, in order. Chance accuracy is 1/8. A solution must store a token because it is a symbol rather than filler, at whatever position it occurs, and keep it for up to a thousand steps; weights that depend only on position cannot do this, since the same position holds filler in one sequence and a symbol in the next. Gu and Dao (2023) use a larger version of the task, with longer sequences and more symbols to copy.

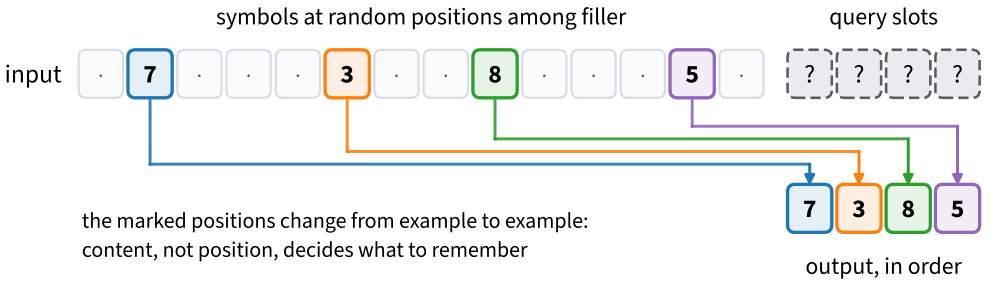


Figure 12.6: The selective copying task. A few marked symbols (color) are scattered among filler tokens (grey) at positions that change from sequence to sequence; at the query tokens, the model must output the symbols in order.

Token 0 is the filler, token 1 the query, and tokens 2 to 9 the symbols. We draw a fresh batch of sequences for every training step, so the model cannot memorize the training data.

```
def copy_batch(batch_size, num_fillers=1024, num_marked=4, num_symbols=8,
               device=None):
    """Symbols at random positions among fillers, then query tokens."""
    X = torch.zeros(batch_size, num_fillers + num_marked, dtype=torch.long,
                    device=device)
    X[:, num_fillers:] = 1  # Query tokens
    Y = torch.randint(2, 2 + num_symbols, (batch_size, num_marked),
                      device=device)
    positions = torch.rand(batch_size, num_fillers, device=device).argsort(1)
    X.scatter_(1, positions[:, :num_marked].sort(1).values, Y)
    return X, Y

X, Y = copy_batch(2, num_fillers=12)
print(X, Y, sep='\n')
```

```
tensor([[0, 2, 0, 0, 0, 0, 6, 0, 0, 5, 0, 7, 1, 1, 1, 1],
        [6, 0, 0, 9, 0, 0, 0, 2, 5, 0, 0, 0, 1, 1, 1, 1]])
tensor([[2, 6, 5, 7],
        [6, 9, 2, 5]])
```

We train every model as `d2l.SeqLM`, the residual stack of Section 12.1, with two blocks of width 48, and read its predictions at the four query positions. Training uses the loop `d2l.train_`-

`queries` of Section 12.1.4.1, with Adam at learning rate 0.003 and gradient clipping, for 2,500 steps of 64 sequences; accuracy is measured on 1,024 new sequences. We start with S4D and the minGRU.

```
def train_copy(model, num_steps=2500, lr=3e-3):
    """Train on fresh batches of selective copying; return test accuracy."""
    device = d2l.try_gpu()
    model.to(device)
    optimizer = torch.optim.Adam(model.parameters(), lr=lr)
    d2l.train_queries(model, lambda step: copy_batch(64, device=device),
                      num_steps, optimizer)
    X, Y = copy_batch(1024, device=device)
    return float(torch.stack([d2l.query_accuracy(model, X_i, Y_i)  # In parts
                              for X_i, Y_i in zip(X.split(128), Y.split(128))
                              ]).mean())
```

```
models = {'S4D': d2l.SeqLM(10, 48, [d2l.S4D(48) for _ in range(2)]),
          'minGRU': d2l.SeqLM(10, 48, [d2l.MinGRU(48, 48) for _ in range(2)])}
for name, model in models.items():
    print(f'{name:8s} accuracy {train_copy(model):.2f}')
```

```
S4D      accuracy 0.19
minGRU   accuracy 0.95
```

S4D answers about one query in five correctly, a little above the chance rate of one in eight: its weights depend on distance, and the same distance holds a symbol in one sequence and a filler in the next. The minGRU answers nine in ten or more in most runs, and rarely fewer than eight. Its decay is chosen by each input, so it can keep a symbol and let fillers pass, and it nearly solves the task. Its gates, however, start near one half: on the delayed recall task of Section 12.2, which needs no selection at all, the minGRU could not keep a symbol across 1,000 fillers unless its gates were initialized with long memories. Selective copying is easier in this respect, because a symbol sometimes sits close to a query, and the short-range gradients from such sequences teach the gates to keep symbols longer and longer.

12.3.2 Selective Dynamics

12.3.2.1 Input-Dependent Step Sizes

Recall the zero-order hold of Section 12.2: a state that obeys $h'(t) = \lambda h(t) + x(t)$ with rate $\lambda < 0$, read at steps of length Δ, decays by $e^{\Delta\lambda}$ per step, and the input enters with weight $(e^{\Delta\lambda}-1)/\lambda$, which is close to Δ when $\Delta|\lambda|$ is small. A small step keeps the state and admits little of the input; a large step forgets the state and admits much. The step size acts as a gate. In S4D it was a learned constant per channel; a selective SSM computes it from the input.

Let the input of the layer have H channels, $\mathbf{x}_t \in \mathbb{R}^H$, and let each channel c have a state $\mathbf{h}_t^{(c)} \in \mathbb{R}^N$ with rates $\lambda_1, \dots, \lambda_N$, initialized to $-1, \dots, -N$ as in S4D. The selective SSM computes three quantities from the input at every step: a step size per channel, and a write vector and a read vector shared by all channels,

$$\Delta_t = \mathrm{softplus}(\mathbf{W}_\Delta \mathbf{x}_t + \mathbf{b}_\Delta), \qquad \mathbf{B}_t = \mathbf{W}_B \mathbf{x}_t, \qquad \mathbf{C}_t = \mathbf{W}_C \mathbf{x}_t, \tag{12.20}$$

where $\text{softplus}(z) = \log(1 + e^z)$ keeps the step sizes positive. The state of each channel then follows the discretized SSM with these step-dependent values,

$$\mathbf{h}_t^{(c)} = e^{\Delta_{t,c}\lambda} \odot \mathbf{h}_{t-1}^{(c)} + \Delta_{t,c}\, x_{t,c}\, \mathbf{B}_t, \qquad y_{t,c} = \mathbf{C}_t^\top \mathbf{h}_t^{(c)} + D_c\, x_{t,c}. \tag{12.21}$$

Here $e^{\Delta_{t,c}\lambda}$ is the vector of decays of the N states, and the input weight is the approximation Δ of the zero-order hold, which Mamba uses. Compared with S4D, three things change, and each is a decision the layer now makes per token. The step size $\Delta_{t,c}$ decides how much channel c forgets and how much of the token it writes; at a filler token the layer can set it near zero and skip the token, and at a symbol it can set it large and store it. The write vector $\mathbf{B}_t$ decides where in the N-dimensional state the token goes, and the read vector $\mathbf{C}_t$ decides what the output reads. Compared with the minGRU, the decay is still chosen by the input, but each channel has N states with rates that span a range, and the step sizes are initialized log-uniformly between 0.001 and 0.1, so that the layer starts with memories from a few steps to a thousand.

12.3.2.2 Training with a Scan

The selective SSM cannot be computed as a convolution: the weight of an input from k steps ago is now a product of k different decays, which depends on the tokens in between, so there is no single kernel. The update Equation 12.21 is still affine in the state, with decays and writes that the layer computes for all positions before the recurrence starts, so `d2l.linear_scan` computes all states in parallel, exactly as for the minGRU.

The scan has a cost that the minGRU did not have: memory. The state at each position now has $H \times N$ numbers, 1,536 for 96 channels with 16 states (the language model below has 16 states per channel), and a minibatch of 64 sequences of 1,024 steps holds 100 million of them, 400 MB in 32-bit floats. Backpropagation through the doubling scan stores every round, $\lceil \log_2 T \rceil = 10$ of them, which takes several gigabytes per layer. A property of linear recurrences avoids this. Let $\mathbf{g}_t$ be the derivative of the loss with respect to the state $\mathbf{h}_t$. The state $\mathbf{h}_t$ affects the loss directly, through the output at step t, and through the next state, $\mathbf{h}_{t+1} = \mathbf{a}_{t+1} \odot \mathbf{h}_t + \mathbf{b}_{t+1}$, so

$$\mathbf{g}_t = \frac{\partial \ell}{\partial \mathbf{h}_t}\Big|_{\text{direct}} + \mathbf{a}_{t+1} \odot \mathbf{g}_{t+1}.$$

This is again a linear recurrence, run backward in time, and a second scan computes it. The gradients of the decays and writes follow directly: $\partial \ell / \partial \mathbf{b}_t = \mathbf{g}_t$ and $\partial \ell / \partial \mathbf{a}_t = \mathbf{g}_t \odot \mathbf{h}_{t-1}$. The backward pass then needs only the decays and the states, not the intermediate rounds. Mamba's fused GPU kernel goes further: it never writes the states to the GPU's main memory at all, but computes them in fast on-chip memory, reads out $\mathbf{C}_t^\top \mathbf{h}_t$ there, and recomputes the states during the backward pass.

In code, the backward pass is a second call of `linear_scan` on the reversed sequence. We give the scan this gradient explicitly, as a custom backward function, and check it against the gradient that automatic differentiation computes through the rounds of the scan.

```
class AdjointScan(torch.autograd.Function):
    """linear_scan whose backward pass is a second scan, run backward."""
    @staticmethod
```

```
        def forward(ctx, a, b):
            h = d2l.linear_scan(a, b)
            ctx.save_for_backward(a, h)
            return h

        @staticmethod
        def backward(ctx, grad_h):
            a, h = ctx.saved_tensors
            # g_t = grad_h_t + a_{t+1} g_{t+1}: a linear recurrence from the end
            a_next = torch.cat([a[:, 1:], torch.ones_like(a[:, :1])], 1)
            g = d2l.linear_scan(a_next.flip(1), grad_h.flip(1)).flip(1)
            h_prev = torch.cat([torch.zeros_like(h[:, :1]), h[:, :-1]], 1)
            return g * h_prev, g

    def adjoint_scan(a, b):
        """All states of h_t = a_t * h_{t-1} + b_t, differentiated by a scan."""
        return AdjointScan.apply(a.expand_as(b), b)
```

```
a = torch.rand(2, 100, 8, requires_grad=True)
b = torch.randn(2, 100, 8, requires_grad=True)
w = torch.randn(2, 100, 8)  # A loss that weights every state
grads_scan = torch.autograd.grad((d2l.linear_scan(a, b) * w).sum(), (a, b))
grads_adjoint = torch.autograd.grad((adjoint_scan(a, b) * w).sum(), (a, b))
deviation = max(float((g1 - g2).abs().max())
                for g1, g2 in zip(grads_scan, grads_adjoint))
print(f'max deviation: {deviation:.1e}')
```

```
max deviation: 9.5e-07
```

The two gradients agree to rounding error. The adjoint scan stores two tensors of the size of the states instead of ten, and in PyTorch it lets the models below train in a few gigabytes.

12.3.3 The Mamba Layer

12.3.3.1 The Layer

A selective SSM mixes information along the sequence, but each channel separately and linearly. The Mamba layer (Gu and Dao 2023) surrounds it with projections, a nonlinearity, and a gate, which mix the channels as the MLP of a transformer block does (Figure 12.7). An input projection widens the d-dimensional input to $H = 2d$ channels and produces a second branch of the same width. A short causal convolution over the last four tokens lets each channel combine a token with its neighbors before the SSM sees it, followed by a SiLU nonlinearity. The SSM output is multiplied by the SiLU of the second branch, a gate, and an output projection returns to width d.

The selective SSM holds the rates of its states, three projections that compute the step sizes and the write and read vectors from the input, and the skip weights D. As in Gu and Dao (2023), the step sizes come from a projection of low rank $R = \lceil H/16 \rceil$, down to R numbers and back up to the H channels: the step sizes of the channels move together, and the projection costs $2HR$ weights instead of H^2. The step sizes are initialized as in S4D: the up-projection starts with small weights, uniform in $\pm R^{-1/2}$, and the biases so that the softplus of the bias alone gives step sizes log-uniformly distributed between `dt_min` and `dt_max`.

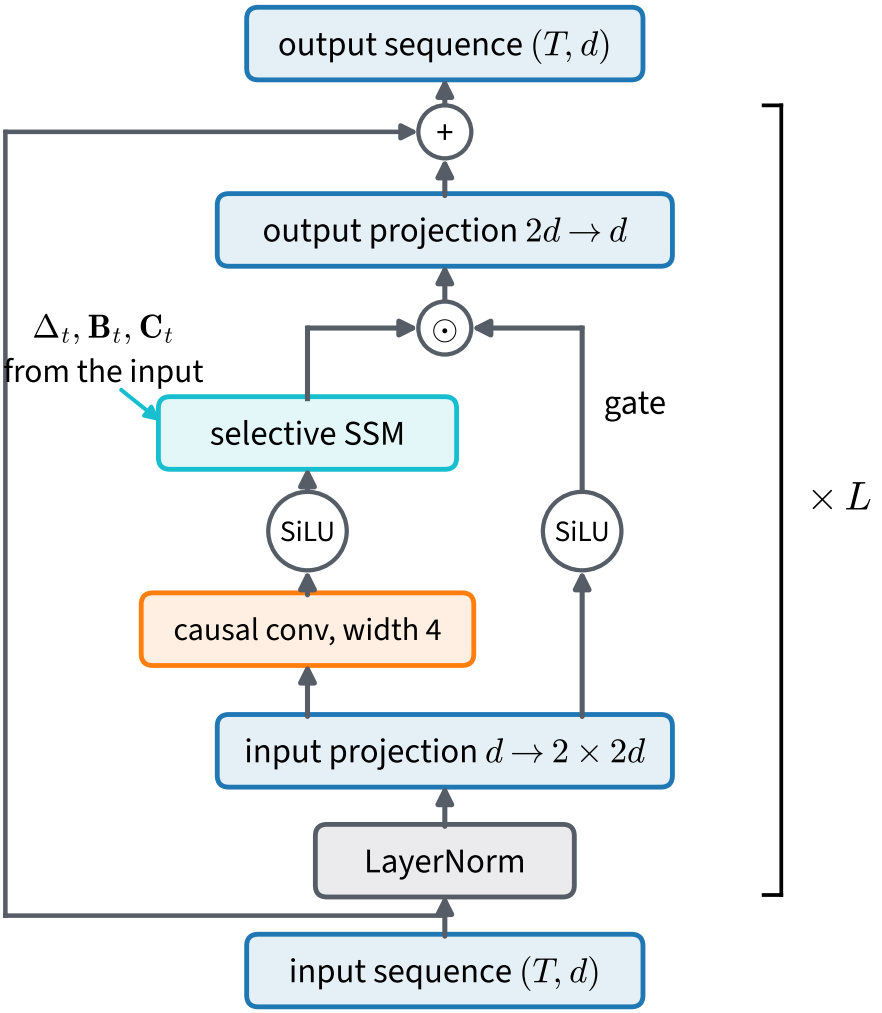


Figure 12.7: The Mamba layer. The input is projected to two branches of width $H = 2d$; one passes through a short causal convolution, a SiLU, and the selective SSM, whose step sizes and write and read vectors are computed from its input; the other gates the result. A final projection returns to width d.

Training then learns how the step sizes depend on the input.

```
class SelectiveSSM(nn.Module):
    """A diagonal SSM whose step sizes and write and read vectors depend on
    the input: N states for each of num_channels channels."""
    def __init__(self, num_channels, num_states=16, dt_min=1e-3, dt_max=1e-1):
        super().__init__()
        H, N, R = num_channels, num_states, math.ceil(num_channels / 16)
        self.log_neg_rate = nn.Parameter(  # Rates -1, ..., -N in every channel
            torch.log(torch.arange(1., N + 1)).repeat(H, 1))
        self.W_dt = nn.Sequential(  # Step size per channel, through rank R
            nn.Linear(H, R, bias=False), nn.Linear(R, H))
        self.W_B = nn.Linear(H, N, bias=False)  # Write vector
        self.W_C = nn.Linear(H, N, bias=False)  # Read vector
        d2l.init_lecun(self)
        dt = dt_min * (dt_max / dt_min) ** torch.rand(H)
        with torch.no_grad():  # Small up-projection, softplus(bias) = dt
            nn.init.uniform_(self.W_dt[1].weight, -R ** -0.5, R ** -0.5)
            self.W_dt[1].bias.copy_(dt + torch.log(-torch.expm1(-dt)))
        self.D = nn.Parameter(torch.ones(H))
```

The method `log_coefficients` computes the step sizes from the input and turns them into the decays $e^{\Delta_{t,c}\lambda_n}$, kept as logarithms, and the writes $\Delta_{t,c}\, x_{t,c}\, \mathbf{B}_t$ of Equation 12.21 for every position, channel, and state. The forward pass scans them and reads every channel's state with $\mathbf{C}_t$.

```
@d2l.add_to_class(SelectiveSSM)
def log_coefficients(self, U):
    """Log-decays and writes, each of shape (batch_size, T, H, N)."""
    dt = F.softplus(self.W_dt(U))  # Step sizes, (batch_size, T, H)
    log_a = dt.unsqueeze(-1) * -torch.exp(self.log_neg_rate)
    return log_a, (dt * U).unsqueeze(-1) * self.W_B(U).unsqueeze(-2)

@d2l.add_to_class(SelectiveSSM)
def forward(self, U):  # U: (batch_size, num_steps, num_channels)
    log_a, b = self.log_coefficients(U)
    H = adjoint_scan(torch.exp(log_a), b)  # States, (batch_size, T, H, N)
    return (H * self.W_C(U).unsqueeze(-2)).sum(-1) + self.D * U
```

The Mamba layer wraps the selective SSM as described above; like the reference implementation, its two projections have no biases. Its interface is that of the other mixers of the chapter, so it can be used in `d2l.SeqLM`. Gu and Dao (2023) stack Mamba layers directly, without the MLP between them; we keep the MLP of `d2l.SeqLM`, so that every mixer of the chapter sits in the same stack.

```
class Mamba(nn.Module):
    """The Mamba layer: a gated selective SSM between two projections."""
    def __init__(self, num_hiddens, num_states=16, expand=2, conv_width=4,
                 **ssm_args):
        super().__init__()
        H = expand * num_hiddens
```

```
        self.W_in = nn.Linear(num_hiddens, 2 * H, bias=False)
        self.conv = nn.Conv1d(H, H, conv_width, groups=H,  # One per channel
                              padding=conv_width - 1)
        self.W_out = nn.Linear(H, num_hiddens, bias=False)
        d2l.init_lecun(self)
        self.ssm = SelectiveSSM(H, num_states, **ssm_args)  # Has its own init

    def ssm_input(self, X):
        """The input of the SSM and the gate branch."""
        U, Z = self.W_in(X).chunk(2, -1)
        U = self.conv(U.transpose(1, 2))[..., :X.shape[1]]  # Keep it causal
        return F.silu(U.transpose(1, 2)), Z

    def forward(self, X):  # X: (batch_size, num_steps, num_hiddens)
        U, Z = self.ssm_input(X)
        return self.W_out(self.ssm(U) * F.silu(Z))
```

12.3.3.2 Selective Copying with Mamba

We train a Mamba model with four states per channel, with the setup above.

```
models['Mamba'] = d2l.SeqLM(10, 48, [Mamba(48, num_states=4)
                                     for _ in range(2)])
print(f"Mamba    accuracy {train_copy(models['Mamba']):.2f}")
```

```
Mamba    accuracy 1.00
```

Mamba answers nearly every query correctly. It combines the two properties that the models of Section 12.2 had separately: decays chosen by the input, as in the minGRU, and memories of many lengths from the start, as in S4D. The minGRU, which has only the first, nearly solves the task; S4D, which has only the second, does not come close.

12.3.3.3 Contributions to the State

Accuracy shows that Mamba keeps the symbols; it does not show how. Because the state is a sum of past writes, each decayed by the decays met since, we can measure how much each input contributes to it. Unrolling the recurrence as in Section 12.2, the state at step T is $\sum_{s \le T} (\prod_{r=s+1}^{T} \mathbf{a}_r) \odot \mathbf{b}_s$, and the size of the term for position s is its contribution. With the logarithms of the decays, each product is the exponential of a difference of cumulative sums.

```
def state_contributions(log_a, b):
    """Size of each position's term in the final state of h = a h + b."""
    cum = log_a.cumsum(1)
    terms = torch.exp(cum[:, -1:] - cum) * b  # Decayed to the last step
    return torch.linalg.vector_norm(terms.flatten(2), dim=-1)  # (batch, T)
```

We apply it to the first layer of the trained S4D and Mamba models, on one test sequence, for the state at the first query token. For Mamba, `log_coefficients` returns the log-decays and

writes at every position. We give S4D the same method, with the decays of Equation 12.18, the same at every step, so that the measurement treats the two layers alike.

```
@d2l.add_to_class(d2l.S4D)
def log_coefficients(self, X):
    """Log-decays and writes of the recurrence, each (batch_size, T, d, N)."""
    rate = -torch.exp(self.log_neg_rate)
    log_a = torch.exp(self.log_dt) * rate  # log a_bar = dt * lambda
    b_bar = torch.expm1(log_a) / rate
    return log_a.expand(*X.shape, -1), b_bar * X.unsqueeze(-1)
```

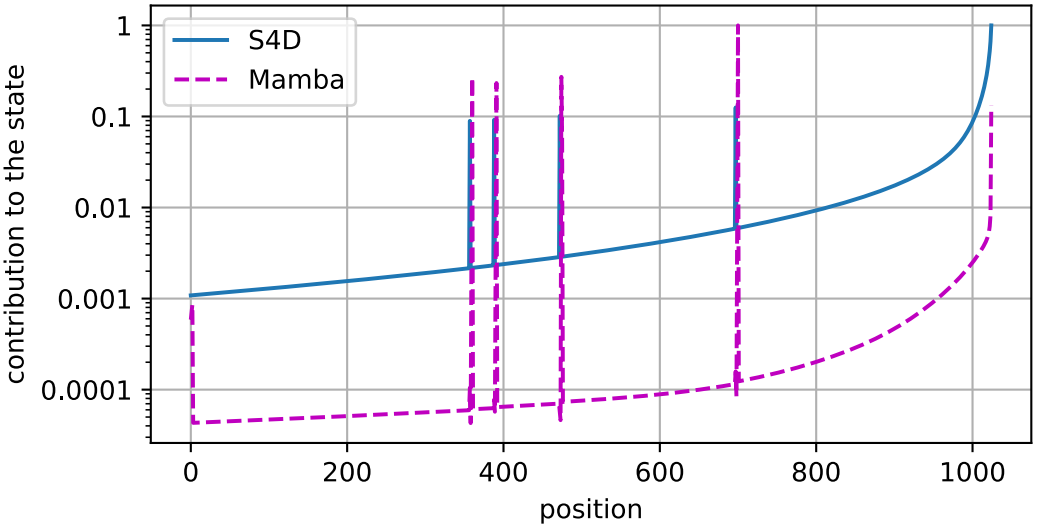
```
X, _ = copy_batch(1, device=d2l.try_gpu())
X = X[:, :-3]  # Up to the first query token
s4d, mamba = models['S4D'], models['Mamba']
with torch.inference_mode():  # The first layer of each model
    c_s4d = state_contributions(
        *s4d.mixers[0].log_coefficients(s4d.first_mixer_input(X)))[0]
    U, _ = mamba.mixers[0].ssm_input(mamba.first_mixer_input(X))
    c_mamba = state_contributions(*mamba.mixers[0].ssm.log_coefficients(U))[0]
```

We also compute the share of the state that comes from the four positions starting at each symbol, which cover less than 2% of the sequence: the short convolution in front of the SSM spreads a token over the next three positions, so the layer can write it at any of them.

```
def symbol_windows(X, width=4):
    """Positions of each symbol and the width - 1 after it, without repeats."""
    starts = (X[0] > 1).nonzero().flatten()  # The four symbols
    offsets = torch.arange(width, device=X.device)
    windows = (starts[:, None] + offsets).flatten()
    windows = windows.unique()  # Windows of nearby symbols may overlap
    return windows[windows < X.shape[1]]  # A window may reach past the end

windows = symbol_windows(X)
for name, c in [('S4D', c_s4d), ('Mamba', c_mamba)]:
    print(f'{name:6s} share from the symbols: {c[windows].sum() / c.sum():.2f}')
d2l.plot(torch.arange(X.shape[1]),
         [(c / c.max()).cpu() for c in (c_s4d, c_mamba)],
         'position', 'contribution to the state', legend=['S4D', 'Mamba'],
         yscale='log', figsize=(6, 3))
```

```
S4D    share from the symbols: 0.03
Mamba  share from the symbols: 0.81
```



For S4D, the contributions rise smoothly toward the query, and the most recent inputs dominate, whatever they are. A symbol may stand out from the fillers next to it, because the size of a write depends on the token, but its decay does not: a symbol hundreds of steps back still contributes far less than the last few fillers. Unless a symbol happens to sit among the last few positions, the windows at the symbols supply only a few percent of the state, about their share of the positions. For Mamba, the contributions spike at the four symbols, far above the fillers around them, wherever the symbols sit, and the windows supply many times their share of the positions: the selective SSM writes the symbols and lets the fillers pass. The task does not determine how much of the state they supply. Runs that copy equally well find different solutions, some reserving nearly the whole state for the symbols, others leaving most of it to the last few positions before the query, so the printed share changes widely from run to run.

12.3.3.4 Language Modeling

We train the Mamba layer as the mixer of `d2l.SeqLM` on the character-level Penn Treebank, with the recipe of the chapter: two blocks of width 128, so that each layer has $H = 256$ channels with 16 states, and 3,000 steps of AdamW through `d2l.fit_lm`. With this recipe the minGRU reached a perplexity of about 3.2 and S4D about 3.35.

```
data = d2l.PennTreebank(batch_size=64, num_steps=128)
```

```
model = d2l.SeqLM(len(data.vocab), 128, [Mamba(128) for _ in range(2)])
print(f'Mamba perplexity: {d2l.fit_lm(model, data):.2f}')
```

```
Mamba perplexity: 3.02
```

Mamba reaches a perplexity of about 3.0, level with the LSTM and the GRU and well below the minGRU and S4D, with fewer parameters than the LSTM model, about 520,000 against 540,000. The low-rank step sizes make the difference: a dense projection of $H \times H$ weights would add about 115,000. A model trained entirely with scans, and generating with a state

of fixed size, models text as well as the gated networks that dominated before transformers.

The state of a selective SSM is an $H \times N$ matrix for every layer, and each step adds to it the outer product of the step-scaled input with the write vector $\mathbf{B}_t$; the read vector $\mathbf{C}_t$ multiplies it. This is a memory of key–value pairs, with $\mathbf{B}_t$ as key and $\mathbf{C}_t$ as query. The next section studies such memories directly: how many pairs they hold, and how to train them with matrix products.

12.3.4 Summary

A state space model with fixed dynamics weights its inputs by their distance, so it cannot keep a token because of what the token is, and it fails at selective copying. A selective state space model computes its step sizes, and the vectors that write to and read from its state, from the current input. A small step size skips a token and a large one stores it, so the step size acts as the gate of a gated recurrent network, while the state keeps N numbers per channel and starts with the multi-timescale initialization of S4D. Mamba solves selective copying over a thousand filler tokens, which the minGRU nearly solves and S4D does not. In the trained model, the few positions after each symbol supply a part of the state at the query many times larger than their share of the positions. The selective model can no longer be computed as a convolution, but its update is still affine in the state, so a scan trains it, and the gradient of the scan is a second scan, run backward. The Mamba layer wraps the selective SSM in projections, a short convolution, and a gate, and it is a strong language model at this scale.

12.3.5 Exercises

1. **A step size is a gate.** Consider one channel with one state, rate $\lambda = -1$, and the exact zero-order hold of Section 12.2, so that $h_t = e^{-\Delta_t} h_{t-1} + (1 - e^{-\Delta_t}) x_t$.
    1. Show that with $\Delta_t = \text{softplus}(s_t)$ the update is $h_t = z_t h_{t-1} + (1 - z_t) x_t$ with $z_t = \sigma(-s_t)$, the blend of a GRU with update gate z_t.
    2. What does the approximation Δ_t of the input weight, which Mamba uses, change? For which Δ_t is it accurate?
2. **Memory of the scan.** For a Mamba layer with H channels and N states, a minibatch of B sequences of length T, and 32-bit numbers, count the bytes that backpropagation stores for the scan, (a) through the rounds of the doubling scan, (b) with the adjoint scan, and (c) with a kernel that stores only the inputs of the layer and recomputes the states. Evaluate the three for $H = 4096$, $N = 16$, $B = 8$, and $T = 8192$, and compare with the memory of a GPU.
3. code **Causality.** The convolution of the Mamba layer must see only the current and earlier tokens; the code pads the sequence so that output t depends on inputs $t - 3, \dots, t$.
    1. Change a token in the middle of a random input sequence and check that the outputs at earlier positions do not change.
    2. Shift the padding so that output t also sees input $t + 1$. What goes wrong, and how does the model's training loss on the copying task react?
4. code **Longer sequences.** Evaluate the Mamba model trained on 1,024 fillers on sequences with 2,048 and 4,096 fillers, without retraining.

1. Does it still copy the symbols? Compare with the minGRU.
2. For each channel of the first layer, compute the decay of its slowest state over 1,000 filler tokens. How many channels can carry a symbol across 4,096 fillers?

5. **Weights that depend on position.** ▯ Consider any model whose state at step T is $\sum_{s \le T} \mathbf{w}_{T-s} \odot f(x_s)$, with fixed weights $\mathbf{w}_k$ and a fixed function f of a single token, and whose output is any function of this state. S4D is of this form. Show that for selective copying with two symbols among fillers, the state cannot depend on the order of the symbols unless the weights $\mathbf{w}_k$ differ for different distances k, and that the model must then distinguish, from the state alone, every pair of positions. Why does this make the task hard for S4D as the number of fillers grows, while a selective model can keep its state unchanged on fillers?

6. code **Which part must be selective?** ▯

   1. Train three variants of the Mamba model on selective copying: with input-dependent step sizes but fixed write and read vectors $\mathbf{B}$ and $\mathbf{C}$, with fixed step sizes but input-dependent $\mathbf{B}_t$ and $\mathbf{C}_t$, and with neither. Which variants solve the task? Relate the result to the contributions to the state measured above.
   2. In the trained Mamba model, set the step sizes of the first layer to their median over filler positions at the four positions after each symbol, so that the layer cannot write the symbols with a large step. How much does the accuracy drop? Repeat with the step sizes set to their median over these positions everywhere else, so that the layer cannot skip the fillers.

7. code **Generation.** Add a method `step` to `SelectiveSSM` and to `Mamba` that processes one token, given the state of the SSM and the last three inputs of the convolution, and returns the output and the new state. Check that stepping through a sequence reproduces the outputs of the forward pass. How many numbers does the layer carry from token to token?

8. code **Short initial memories.** Train the Mamba model with step sizes initialized between 0.5 and 1 (`dt_min=0.5, dt_max=1.0`), so that every state starts with a memory of a few steps.

   1. Does it learn to copy the symbols within the training budget? If so, plot the step sizes of the first layer on filler tokens before and after training, and explain how the model could have learned long memories from gradients that, at the start, vanish over a thousand steps. (Hint: where do the symbols closest to the queries sit?)
   2. Repeat with 4,096 fillers, and with the minGRU initialized as in the long-range recall experiment of Section 12.2.

12.4 Matrix-State Recurrences

The selective models of Section 12.3 decide, token by token, what to keep and for how long. Much of what a language model must remember comes in pairs, a name together with what it stands for. A program defines `rate = 0.05` near its top and uses `rate` two hundred lines later. A story introduces a character as "the engineer" and refers to "the engineer" again three pages on. To predict such a token, a model must store a *pair* and later retrieve the second half of the pair when the first half reappears. This is *associative recall*, and transformers do it with attention: the current token's query is compared with the keys of all earlier tokens,

and the values of the best matches are returned. Can a recurrent state of fixed size store pairs too, and how many?

This section makes the state a matrix that stores key–value pairs. A pair is written by adding the outer product of its value and its key; it is read by multiplying the matrix by a query. This is linear attention, the variant of attention in Section 10.5 that drops the softmax, and, up to its decay, it is also what the selective state space model of Section 12.3 computes, with different names for the same vectors. We ask how many pairs such a memory can hold, how a decay keeps it from filling up, and how to train it fast: a decayed matrix recurrence can be computed as masked attention, as a scan, or in chunks that combine the two. The section ends with a limit that the next section removes: every write adds, so a memory cannot change what it has stored.

12.4.1 An Associative Memory

12.4.1.1 Writing and Reading Pairs

The simplest memory for pairs is a sum: it adds each new pair to what it holds and reads a value back with its key. A small example shows how this works and where it fails. Start with two keys in $\mathbb{R}^2$, $\mathbf{k}_1 = (1,0)^\top$ and $\mathbf{k}_2 = (0,1)^\top$, with the values $v_1 = 3$ and $v_2 = 5$. Store both in one row vector, the sum of each value times its key, $\mathbf{S} = v_1\mathbf{k}_1^\top + v_2\mathbf{k}_2^\top = (3,5)$. Multiplying by a key reads its value back: $\mathbf{S}\mathbf{k}_1 = 3$ and $\mathbf{S}\mathbf{k}_2 = 5$. Now store a third pair, $\mathbf{k}_3 = (1,1)^\top/\sqrt{2}$ with $v_3 = 7$. The memory becomes $(3{+}7/\sqrt{2}, 5{+}7/\sqrt{2}) \approx (7.95, 9.95)$, and reading $\mathbf{k}_1$ now returns 7.95 instead of 3. Two dimensions hold two pairs exactly; a third key must overlap with the others, and its value leaks into theirs.

With values $\mathbf{v}_t \in \mathbb{R}^{d_v}$ and keys $\mathbf{k}_t \in \mathbb{R}^{d_k}$, the memory is a matrix $\mathbf{S}_t \in \mathbb{R}^{d_v \times d_k}$, the transpose of the layout in Section 10.5. The key width d_k is the d of Chapter 10; the subscript keeps it apart from d_v, since both widths appear in the shape of the memory. Each step writes one pair and the output reads the memory with a query $\mathbf{q}_t \in \mathbb{R}^{d_k}$:

$$\mathbf{S}_t = \mathbf{S}_{t-1} + \mathbf{v}_t\mathbf{k}_t^\top, \qquad \mathbf{y}_t = \mathbf{S}_t\mathbf{q}_t. \tag{12.22}$$

The outer product $\mathbf{v}_t\mathbf{k}_t^\top$ is a matrix of rank one that maps the direction $\mathbf{k}_t$ to $\mathbf{v}_t$; the state is the sum of these maps. The keys, values, and queries are linear projections of the input, as in attention, so the network learns what to store under which key and what to look up. A weight matrix that the network rewrites at every step is called a *fast weight*; Schlag et al. (2021) trace linear attention back to this older idea.

12.4.1.2 Linear Attention

The recurrence Equation 12.22 does not look like attention, yet it computes a form of it. Unrolling it from $\mathbf{S}_0 = \mathbf{0}$ gives

$$\mathbf{y}_t = \Big(\sum_{s\le t} \mathbf{v}_s\mathbf{k}_s^\top\Big)\mathbf{q}_t = \sum_{s\le t} (\mathbf{k}_s^\top\mathbf{q}_t)\,\mathbf{v}_s.$$

The right side is causal attention with the score $\mathbf{k}_s^\top\mathbf{q}_t$ and no softmax: every earlier value, weighted by how well its key matches the query. This is the *linear attention* of Equation 10.30 in Section 10.5, with the identity as the feature map and without the normalizer (the models

of this chapter normalize the output of the layer instead): because the score is a product of features rather than an exponential, the sum folds into a fixed-size matrix, and the cost per token does not grow with the length of the context (Katharopoulos et al. 2020). The left side is the recurrence. The same computation is a transformer layer read one way and a recurrent network read the other.

12.4.1.3 Interference

How many pairs can $\mathbf{S}$ hold? Store n pairs, $\mathbf{S} = \sum_{i=1}^{n} \mathbf{v}_i \mathbf{k}_i^\top$, with keys of unit length, and read key j:

$$\mathbf{S}\mathbf{k}_j = \mathbf{v}_j + \sum_{i \neq j} (\mathbf{k}_i^\top \mathbf{k}_j)\, \mathbf{v}_i. \tag{12.23}$$

The read returns the stored value plus every other value, weighted by the overlap of its key with the target key. Only mutually orthogonal keys are read back without error, and $\mathbb{R}^{d_k}$ holds at most d_k of them. For more pairs, or for keys that are not orthogonal, interference is unavoidable. How large it is depends on the keys; for random keys we can compute it.

Proposition. Let the keys $\mathbf{k}_1, \dots, \mathbf{k}_n$ be independent and uniformly distributed on the unit sphere of $\mathbb{R}^{d_k}$, and let the values be unit vectors chosen independently of the keys. Then the expected squared read error at every stored key is

$$\mathbb{E}\left\|\mathbf{S}\mathbf{k}_j - \mathbf{v}_j\right\|^2 = \frac{n-1}{d_k}. \tag{12.24}$$

Proof. The error is $\sum_{i \neq j} (\mathbf{k}_i^\top \mathbf{k}_j)\, \mathbf{v}_i$. Its squared norm is a double sum over i and l. For $i \neq l$, the expected cross term vanishes: given $\mathbf{k}_j$, the overlaps $\mathbf{k}_i^\top \mathbf{k}_j$ and $\mathbf{k}_l^\top \mathbf{k}_j$ are independent with mean zero. What remains is $\sum_{i \neq j} \mathbb{E}[(\mathbf{k}_i^\top \mathbf{k}_j)^2]$, using $\|\mathbf{v}_i\| = 1$. A random unit vector has the same expected squared coordinate in every direction, and these d_k expectations sum to $\|\mathbf{k}_i\|^2 = 1$, so each term is $1/d_k$. There are $n-1$ terms. ■

The error grows in proportion to the number of stored pairs and falls in proportion to the key dimension. Once n approaches d_k, the interference is as large as the signal, whose squared norm is one. Time does not appear: a pair written long ago is read back as well as one written a moment ago: memories are lost because other pairs interfere with them, not because they are old.

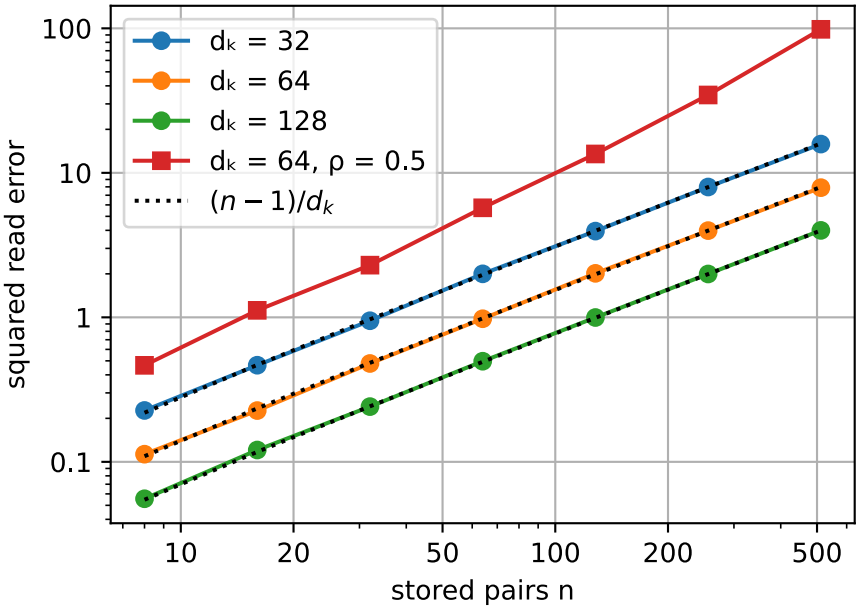
We check the proposition by storing random pairs and reading them back. The values come from a codebook of 256 random unit vectors, so we can also ask whether the nearest codebook entry to the read is the stored one. A last curve breaks the assumptions: every key shares a common direction, with weight $\rho = 0.5$, so the keys are correlated. The computation needs only NumPy.

```
def matrix_memory(d_k, nums, rho=0, num_values=256, trials=20, seed=0):
    """Read error and recall of S = sum_i v_i k_i^T over random unit keys."""
    rng = np.random.default_rng(seed)
    unit = lambda X: X / np.linalg.norm(X, axis=-1, keepdims=True)
    codebook = unit(rng.standard_normal((num_values, d_k)))
    errors, recalls = [], []
```

```
    for n in nums:
        error = recall = 0
        for _ in range(trials):
            K = unit(rng.standard_normal((n, d_k)))  # Keys as rows
            K = unit(np.sqrt(1 - rho**2) * K + rho * unit(rng.standard_normal(d_k)))
            ids = rng.integers(0, num_values, n)
            S = codebook[ids].T @ K  # Sum of v_i k_i^T
            reads = K @ S.T  # Row j is S k_j
            error += ((reads - codebook[ids]) ** 2).sum(1).mean() / trials
            recall += ((reads @ codebook.T).argmax(1) == ids).mean() / trials
        errors.append(error)
        recalls.append(recall)
    return errors, recalls
```

```
nums = [8, 16, 32, 64, 128, 256, 512]
results = {f'd  = {d_k}': matrix_memory(d_k, nums) for d_k in (32, 64, 128)}
results['d  = 64,   = 0.5'] = matrix_memory(64, nums, rho=0.5)
predictions = [[(n - 1) / d_k for n in nums] for d_k in (32, 64, 128)]
d2l.plot(nums, [err for err, _ in results.values()] + predictions,
         'stored pairs n', 'squared read error',
         legend=list(results) + ['$(n-1)/d_k$'], xscale='log', yscale='log',
         fmts=('-o', '-o', '-o', '-s', 'k:', 'k:', 'k:'), figsize=(5, 3.5))
print('recall at n =    ', ' '.join(f'{n:4d}' for n in nums))
for name, (_, recalls) in results.items():
    print(f'{name:17s}', ' '.join(f'{r:4.2f}' for r in recalls))
```

```
recall at n =        8   16   32   64  128  256  512
d  = 32           1.00 0.99 0.96 0.74 0.39 0.16 0.07
d  = 64           1.00 1.00 1.00 1.00 0.96 0.72 0.37
d  = 128          1.00 1.00 1.00 1.00 1.00 1.00 0.94
d  = 64,   = 0.5  1.00 0.99 0.83 0.49 0.21 0.07 0.03
```



The measured errors lie on the predicted lines $(n-1)/d_k$ at all three widths. The recall shows what this means for retrieval: with $d_k = 64$, every pair is recovered up to $n = 64$, 96% at $n = 128$, 72% at $n = 256$, and 37% at $n = 512$, and doubling d_k doubles the number of pairs held at a given recall. Correlated keys do worse. With a shared component of weight $\rho = 0.5$, the error is about four times larger already at $n = 8$, it grows faster than in proportion to n, and only half of 64 pairs are recovered. What a matrix memory can hold depends on how different its keys are, not only on how many there are.

12.4.2 Decay

12.4.2.1 Forgetting Old Pairs

Linear attention never forgets. After n writes all n pairs share the memory, and by Equation 12.24 the read error grows with n, however old most of the pairs are. A decay before each write shrinks the old pairs,

$$\mathbf{S}_t = a_t\,\mathbf{S}_{t-1} + \mathbf{v}_t\mathbf{k}_t^\top, \qquad a_t \in (0,1), \tag{12.25}$$

so that a pair written k steps ago has weight a^k if the decay is constant. The weights of all past pairs then sum to $1/(1-a)$, and the memory behaves as if it held about the last $1/(1-a)$ pairs: 10 for $a = 0.9$, 100 for $a = 0.99$. As in Section 12.2, a decay that depends on the input lets the model choose: hold the memory through a paragraph and clear it at a new document. Such a decay is the first of the two changes that made linear attention competitive in practice (Section 10.5); the second, keeping a few softmax layers for recall, is the subject of Section 12.6.

The decay can be shaped in several ways, each a one-line change of Equation 12.25. RetNet (Sun et al. 2023) uses a fixed scalar per head, so that different heads remember for different lengths. Mamba-2 (Dao and Gu 2024) uses an input-dependent scalar per head, as above. Gated linear attention (Yang, Wang, Shen, et al. 2024) decays each key coordinate at its own

rate, $\mathbf{S}_t = \mathbf{S}_{t-1}\mathrm{diag}(\alpha_t) + \mathbf{v}_t\mathbf{k}_t^\top$, so that some directions of the key space are kept longer than others.

The selective state space model of Section 12.3 is a matrix memory of the same kind. Its state holds N numbers for each of its H channels, an $H \times N$ matrix $\mathbf{S}_t$. Each step adds the outer product of the channel inputs, scaled by the step sizes, $\Delta_t \odot \mathbf{x}_t$, with the vector $\mathbf{B}_t \in \mathbb{R}^N$, and the output reads $\mathbf{S}_t\mathbf{C}_t$. In the terms of this section the value is $\Delta_t \odot \mathbf{x}_t$, the key is $\mathbf{B}_t$, the query is $\mathbf{C}_t$, and $d_k = N$. Its decay is elementwise, a different rate for every entry of $\mathbf{S}_t$. Mamba-2 replaced it by one scalar per head, which loses some flexibility and gains the matrix products of the dual form below.

12.4.3 The Dual Form

12.4.3.1 Unrolling with Decay

Written as a recurrence, the decayed memory processes one token at a time. Without decay, linear attention can also be computed as attention, over all positions at once (Section 10.5); we ask whether this survives the decay. With a scalar decay, unrolling Equation 12.25 from $\mathbf{S}_0 = \mathbf{0}$ gives each output as a weighted sum over the past, in which a pair written at step s is scaled by all decays met since:

$$\mathbf{y}_t = \sum_{s \le t} L_{ts}\,(\mathbf{k}_s^\top \mathbf{q}_t)\,\mathbf{v}_s, \qquad L_{ts} = a_{s+1}a_{s+2}\cdots a_t.$$

Stack the queries, keys, values, and outputs of a sequence as the rows of $\mathbf{Q}$, $\mathbf{K}$, $\mathbf{V}$, and $\mathbf{Y}$, and set $L_{ts} = 0$ for $s > t$. Then all outputs together are

$$\mathbf{Y} = (\mathbf{L} \odot \mathbf{Q}\mathbf{K}^\top)\,\mathbf{V}. \tag{12.26}$$

This is masked attention. With every $a_t = 1$, $\mathbf{L}$ is the causal mask of a transformer, ones on and below the diagonal, and Equation 12.26 is causal linear attention. With decays, $\mathbf{L}$ is a *learned, decaying* mask: a decayed linear recurrence is attention in which older tokens count less, by an amount the inputs choose. Dao and Gu (2024) call this correspondence *state space duality*. It holds for linear attention, not for softmax attention, whose exponential does not factor this way.

The two sides of the duality are two orders of computing the same products. Forming $\mathbf{L} \odot \mathbf{Q}\mathbf{K}^\top$ first is attention: all positions in parallel, at a cost and memory quadratic in T. Summing from the left, one step at a time, is the recurrence: the partial sums are the states $\mathbf{S}_t$, the cost is linear in T, and the steps are sequential. We check that both give the same outputs, starting with the recurrence.

```
def linear_attention_recurrent(Q, K, V, a):
    """y_t = S_t q_t with S_t = a_t S_{t-1} + v_t k_t^T, step by step; a_t
    may also hold one decay per key coordinate, S_t = S_{t-1} diag(a_t) + ..."""
    S, outputs = Q.new_zeros(V.shape[-1], K.shape[-1]), []
    for q, k, v, a_t in zip(Q, K, V, a):
        S = a_t * S + torch.outer(v, k)
        outputs.append(S @ q)
    return torch.stack(outputs)
```

The dual form needs the mask $\mathbf{L}$. Its entries are products of decays, which we compute from the logarithms of the decays: $\log L_{ts}$ is the sum of $\log a_r$ over $s < r \leq t$, a difference of two cumulative sums. Above the diagonal these differences are positive and their exponentials can overflow, so we set them to $-\infty$ before exponentiating, which turns them into exact zeros. We save the mask, because the chunked algorithms below and in the next section use it too.

```
def decay_mask(log_a):
    """L[t, s] = a_{s+1} * ... * a_t for s <= t, and 0 for s > t."""
    cum = log_a.cumsum(-1)
    log_L = cum[..., :, None] - cum[..., None, :]  # Sum of log a over (s, t]
    causal = torch.ones(log_L.shape[-2:], dtype=torch.bool,
                        device=log_a.device).tril()
    return torch.exp(log_L.masked_fill(~causal, -torch.inf))

def linear_attention_dual(Q, K, V, a):
    """The same outputs as Y = (L * Q K^T) V, all positions at once."""
    return (decay_mask(torch.log(a)) * (Q @ K.T)) @ V
```

We compare the two on a random sequence of 512 steps with decays between 0.8 and 0.99.

```
gen = torch.Generator().manual_seed(0)  # Local: later training stays unseeded
T, d_k, d_v = 512, 64, 64
Q, K, V = (torch.randn(T, d, generator=gen) for d in (d_k, d_k, d_v))
a = 0.8 + 0.19 * torch.rand(T, generator=gen)
Y = linear_attention_recurrent(Q, K, V, a)
Y_dual = linear_attention_dual(Q, K, V, a)
print(f'relative deviation: {(Y_dual - Y).abs().max() / Y.abs().max():.1e}')
```

```
relative deviation: 1.5e-06
```

The two orders agree to rounding error. Neither is what we want for training: the recurrence is sequential, and the dual form stores a $T \times T$ matrix, 64 MB per head in 32-bit floats at $T = 4{,}096$.

12.4.3.2 Chunked Computation

Chunking combines the two, as Equation 10.32 of Section 10.5 did for linear attention without decay: the quadratic form within each chunk, and a state carried from chunk to chunk. The mask shows where the decays enter. Cut the sequence into chunks of m steps and partition the matrix $\mathbf{L} \odot \mathbf{Q}\mathbf{K}^\top$ into $m \times m$ blocks, as in Figure 12.8. The blocks on the diagonal are small masked attention products, computed in parallel. The blocks below the diagonal are never formed: everything a chunk needs from the entire past is the $d_v \times d_k$ state at the chunk's start. With decays, the chunked computation has three parts:

1. *Within chunks*: the dual form Equation 12.26 on each chunk.
2. *Across chunks*: each chunk's writes, decayed to the chunk's end, form one matrix; the states at the chunk boundaries follow from a linear recurrence over the T/m chunks, $\mathbf{S}^{(c)} = (\text{decay over chunk } c)\,\mathbf{S}^{(c-1)} + (\text{writes of chunk } c)$, which `linear_scan` of Section 12.2 computes with matrix-valued states.
3. *Reading the past*: each position adds the read of the state at its chunk's start, decayed to its own position.

Almost all of the work is in matrix products, which GPUs do well, and the memory holds $m \times m$ blocks and one state per chunk instead of a $T \times T$ matrix. Mamba-2 and the models of the next section train this way.

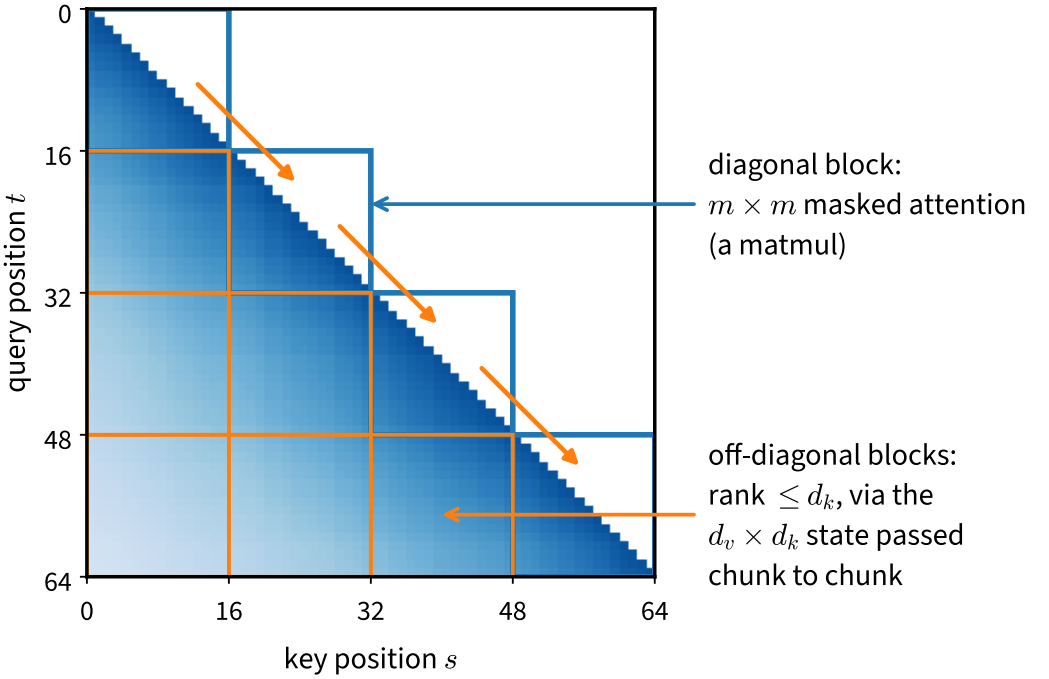


Figure 12.8: The mask $\mathbf{L}$ of Equation 12.26 for random decays, partitioned into chunks. Diagonal blocks (blue) are small masked-attention products computed in parallel. Every block below the diagonal (orange) is accounted for by the state passed from chunk to chunk (arrows); the shading fades as the decays compound.

A decay per key coordinate, $\mathbf{S}_t = \mathbf{S}_{t-1}\mathrm{diag}(\alpha_t) + \mathbf{v}_t\mathbf{k}_t^\top$, as in gated linear attention, changes only the first part. Let c_{tj} be the sum of $\log \alpha_{rj}$ from the chunk's first step to step t. The decayed score of query t and key $s \leq t$ is then $\sum_j q_{tj}k_{sj}\exp(c_{tj}-c_{sj})$: every key coordinate has a mask of its own, and no single mask $\mathbf{L}$ multiplies $\mathbf{QK}^\top$. `decay_mask` computes the d_k masks at once, $d_k m^2$ numbers per chunk.

It is tempting to avoid them. Since $\exp(c_t - c_s) = \exp(c_t)\exp(-c_s)$, scaling each query by $\exp(c_t)$ and each key by $\exp(-c_s)$ turns the masked product of a chunk into a plain causal one. In floating point this fold breaks. The factors $\exp(-c_s)$ grow as the decays of a chunk compound and overflow 32-bit floats once those multiply to less than e^{-88}, and learned gates get there: in gated linear attention on the recall task of Section 12.6, the largest such exponent grows from about 20 after 2,000 training steps to between 45 and 75 after 10,000, and two of three runs of 32,000 steps overflowed. One overflow turns the gradients, and then every weight, into NaN, and the run sits at chance from then on without raising any error. The function below never forms such factors. Every exponential it computes is the decay over a stretch of time, $\exp(c_t - c_s)$ with $s \leq t$, $\exp(c_t)$, or $\exp(c_m - c_s)$, and lies between 0 and 1, however strong the decays get.

The function takes a batch of sequences and the logarithms of the decays, one per step or

one per key coordinate. Its three parts follow the list above; the decay from step t to the end of its chunk is the exponential of a difference of the cumulative log decays `cum`.

```
def chunked_linear_attention(Q, K, V, log_a, chunk_size):
    """y_t = S_t q_t with S_t = S_{t-1} diag(a_t) + v_t k_t^T, in chunks.

    Q, K: (batch_size, T, d_k); V: (batch_size, T, d_v); log_a: (batch_size,
    T, d_k) for a decay per key coordinate, or (batch_size, T) for one per step
    """
    B, T, m = Q.shape[0], Q.shape[1], chunk_size
    if log_a.dim() == 2:  # The same decay for every key coordinate
        log_a = log_a.unsqueeze(-1)
    Q, K, V, log_a = (X.reshape(B, T // m, m, -1) for X in (Q, K, V, log_a))
    # 1. Within chunks: the dual form, with one mask per key coordinate
    L = decay_mask(log_a.mT)  # (B, T / m, 1 or d_k, m, m), entries <= 1
    if L.shape[2] == 1:  # One mask for all coordinates: a masked Q K^T
        Y = (L[:, :, 0] * (Q @ K.mT)) @ V
    else:
        Y = (L * Q.mT.unsqueeze(-1) * K.mT.unsqueeze(-2)).sum(2) @ V
    # 2. Across chunks: the writes of each chunk, decayed to its end, and
    #    a scan over chunks for the state at every chunk's end
    cum = log_a.cumsum(2)  # Log decay from the chunk's start to each step
    writes = V.mT @ (K * torch.exp(cum[:, :, -1:] - cum))
    S = d2l.linear_scan(torch.exp(cum[:, :, -1:]), writes)
    S = torch.cat([torch.zeros_like(S[:, :1]), S[:, :-1]], 1)  # At the start
    # 3. Every step reads the state at its chunk's start, decayed to it
    return (Y + (Q * torch.exp(cum)) @ S.mT).reshape(B, T, -1)
```

The chunked outputs match the recurrence for every chunk size, with one decay per step and with one per key coordinate.

```
a_key = 0.8 + 0.19 * torch.rand(T, d_k,  # One decay per key coordinate
                                generator=gen)
decays = {'per step': (a, Y),
          'per key': (a_key, linear_attention_recurrent(Q, K, V, a_key))}
for m in [16, 64, 256]:
    for name, (decay, Y_ref) in decays.items():
        Y_chunked = chunked_linear_attention(Q[None], K[None], V[None],
                                             torch.log(decay)[None], m)[0]
        print(f'chunk size {m:3d}, decay {name}: relative deviation '
              f'{(Y_chunked - Y_ref).abs().max() / Y_ref.abs().max():.1e}')
```

```
chunk size  16, decay per step: relative deviation 2.2e-07
chunk size  16, decay per key: relative deviation 2.1e-07
chunk size  64, decay per step: relative deviation 2.5e-07
chunk size  64, decay per key: relative deviation 2.1e-07
chunk size 256, decay per step: relative deviation 8.8e-07
chunk size 256, decay per key: relative deviation 6.4e-07
```

How fast are the three orders? We time them on one sequence of 4,096 steps on a GPU.

```
device = d2l.try_gpu()
T = 4096
Q, K, V = (torch.randn(T, d, device=device) for d in (d_k, d_k, d_v))
a = 0.8 + 0.19 * torch.rand(T, device=device)
forms = {'recurrence': lambda: linear_attention_recurrent(Q, K, V, a),
         'dual': lambda: linear_attention_dual(Q, K, V, a)}
for m in [16, 64, 256]:
    forms[f'chunks of {m}'] = lambda m=m: chunked_linear_attention(
        Q[None], K[None], V[None], torch.log(a)[None], m)
for name, f in forms.items():
    print(f'{name:14s} {1e3 * d2l.Benchmark(f, warmup=1, repeats=5).time:8.2f} ms')
```

```
recurrence       258.73 ms
dual               2.09 ms
chunks of 16       1.26 ms
chunks of 64       1.29 ms
chunks of 256      1.13 ms
```

The recurrence is the slowest by two orders of magnitude, because its 4,096 steps run one after another. At this length the dual form and the chunked form take similar times, about a millisecond or less. They differ in memory: the dual form stores the $T \times T$ matrix of weights, 64 MB per head in 32-bit floats, which grows with the square of the length, while chunks of 64 store $T \times 64$ weights and one 64×64 state per chunk, about 2 MB, which grows in proportion to the length.

12.4.4 A Gated Linear Attention Layer

12.4.4.1 The Layer

A layer built on the decayed matrix memory computes, for every head, a query, a key, a value, and decays from the input, runs the chunked recurrence, and projects the outputs of all heads back to the model width. We call it *gated linear attention*, after its decay gate, and build it as Yang, Wang, Shen, et al. (2024) and the layers that followed it do. Four parts surround the memory; the experiments below and in Section 12.6 measure what they contribute:

- *A short convolution.* The projected queries, keys, and values pass through a causal convolution of width 4, one filter per channel, and a SiLU. Each of them can then draw on the few tokens before it, so that a pair spread over neighboring tokens, a name followed by its value, can be written as one key and one value.
- *Unit queries and keys,* the setting of Equation 12.24.
- *A decay per key coordinate,* $\alpha_t = \sigma(\mathbf{W}_2\mathbf{W}_1\mathbf{x}_t + \mathbf{b})^{1/16}$, computed through a projection of rank 16. The exponent $1/16$ keeps the decays close to one while the gate moves, and the biases start the coordinates with memories of 4 to 256 steps, spread log-uniformly, as the timescales of Section 12.2 were.
- *An output norm and gate.* The output of each head is normalized to unit root mean square, which keeps its scale independent of how many pairs the memory holds, and multiplied by a gate $\mathrm{SiLU}(\mathbf{W}_g\mathbf{x}_t)$ before the output projection.

```
class GatedLinearAttention(nn.Module):
    """Multi-head linear attention with a gated decay per key coordinate."""
    gate_scale = 16  # log a = log sigmoid(z) / 16 keeps decays near one
```

```
    def __init__(self, num_hiddens, num_heads, chunk_size=16, conv_width=4,
                 timescales=(4, 256)):
        super().__init__()
        d = num_hiddens
        self.num_heads, self.chunk_size = num_heads, chunk_size
        self.W_qkv = nn.Linear(d, 3 * d, bias=False)
        self.conv = nn.Conv1d(3 * d, 3 * d, conv_width, groups=3 * d,
                              padding=conv_width - 1)  # One filter per channel
        self.W_a = self.decay_gate(d)
        self.W_g = nn.Linear(d, d)  # Output gate
        self.W_o = nn.Linear(d, d, bias=False)
        d2l.init_lecun(self)
        self.init_decays(*timescales)  # Memories of 4 to 256 steps by default

    def decay_gate(self, d, rank=16):
        """Low-rank projection to one gate per key coordinate."""
        return nn.Sequential(nn.Linear(d, rank, bias=False), nn.Linear(rank, d))

    def init_decays(self, tau_min, tau_max):
        """Start the gates at decays exp(-1/tau), tau log-uniform in range."""
        b = self.W_a[-1].bias
        tau = torch.logspace(math.log10(tau_min), math.log10(tau_max), len(b))
        with torch.no_grad():  # log sigmoid(b) / gate_scale = -1 / tau
            b.copy_(-torch.log(torch.expm1(self.gate_scale / tau)))
```

The recurrence sits in a method of its own, `memory`, and the decays in `decays`, so that the next section can replace them. Both see the heads side by side: `heads_to_batch` moves the heads of a minibatch into its batch axis, so that the chunked computation treats every head as a sequence of its own, and `batch_to_heads` moves them back.

```
def heads_to_batch(Z, num_heads):
    """(batch_size, T, num_heads * e) -> (batch_size * num_heads, T, e)."""
    B, T, _ = Z.shape
    return Z.reshape(B, T, num_heads, -1).transpose(1, 2).flatten(0, 1)

def batch_to_heads(Y, batch_size):
    """(batch_size * num_heads, T, e) -> (batch_size, T, num_heads * e)."""
    _, T, e = Y.shape
    return Y.reshape(batch_size, -1, T, e).transpose(1, 2).flatten(2)

@d2l.add_to_class(GatedLinearAttention)
def decays(self, X):
    """Log decays, (batch_size * num_heads, T, d_k)."""
    return heads_to_batch(F.logsigmoid(self.W_a(X)) / self.gate_scale,
                          self.num_heads)

@d2l.add_to_class(GatedLinearAttention)
def memory(self, X, Q, K, V, log_a):
    """Outputs of the decayed matrix memory of every head. X is unused here;
    the delta rule of the next section reads its write strengths from it."""
    return chunked_linear_attention(Q, K, V, log_a, self.chunk_size)
```

The forward pass runs the projections through the convolution and the SiLU, splits them into the queries, keys, and values of the heads, scales queries and keys to unit length, runs the memory, normalizes and gates the output of each head, and projects the concatenated heads back to the model width.

```
@d2l.add_to_class(GatedLinearAttention)
def forward(self, X):  # X: (batch_size, num_steps, num_hiddens)
    B, T = X.shape[:2]
    QKV = self.conv(self.W_qkv(X).mT)[..., :T].mT  # Causal: drop the tail
    Q, K, V = (heads_to_batch(Z, self.num_heads)
               for Z in F.silu(QKV).chunk(3, -1))
    Q, K = F.normalize(Q, dim=-1), F.normalize(K, dim=-1)  # Unit length
    Y = self.memory(X, Q, K, V, self.decays(X))  # (B * num_heads, T, d / h)
    Y = F.rms_norm(Y, Y.shape[-1:], eps=1e-6)  # Unit RMS per head
    return self.W_o(batch_to_heads(Y, B) * F.silu(self.W_g(X)))
```

12.4.4.2 Language Modeling

We train the layer with four heads of dimension 32 as the mixer of `d2l.SeqLM`, on the character-level Penn Treebank with the recipe of Section 12.1 (`d2l.fit_lm`). Each head's state holds 32×32 numbers, so a layer holds 4,096, against 2,048 for the S4D layer of Section 12.2 with 16 states per channel.

```
data = d2l.PennTreebank(batch_size=64, num_steps=128)
```

```
model = d2l.SeqLM(len(data.vocab), 128,
                  [GatedLinearAttention(128, num_heads=4) for _ in range(2)])
print(f'gated linear attention perplexity: {d2l.fit_lm(model, data):.2f}')
```

```
gated linear attention perplexity: 3.12
```

The model reaches a perplexity of about 3.1, better than the minGRU (3.2) and S4D (3.35), behind the LSTM, the GRU, and Mamba (about 3.0). How much of this comes from the parts around the memory? We remove the short convolution, the SiLU, and the output gate, and keep the unit queries and keys, the decays, and the output norm.

```
class PlainLinearAttention(GatedLinearAttention):
    """Gated linear attention without the short convolution, the SiLU, and
    the output gate."""
    def forward(self, X):
        Q, K, V = (heads_to_batch(Z, self.num_heads)
                   for Z in self.W_qkv(X).chunk(3, -1))
        Q, K = F.normalize(Q, dim=-1), F.normalize(K, dim=-1)
        Y = self.memory(X, Q, K, V, self.decays(X))
        Y = F.rms_norm(Y, Y.shape[-1:], eps=1e-6)
        return self.W_o(batch_to_heads(Y, X.shape[0]))
```

```
model = d2l.SeqLM(len(data.vocab), 128,
                  [PlainLinearAttention(128, num_heads=4) for _ in range(2)])
print(f'without convolution, SiLU, and gate: {d2l.fit_lm(model, data):.2f}')
```

```
without convolution, SiLU, and gate: 3.19
```

Without them, the same layer reaches only about 3.2. The matrix memory itself does not help much on this task, as the models of Section 12.2 suggested: predicting the next character rarely requires looking up a pair stored a hundred characters earlier. The tasks of Section 12.6 do, and there the size of the memory limits how much it can recall.

One limitation is built into every model in this section. A write adds $\mathbf{v}_t\mathbf{k}_t^\top$ to the memory, whatever the memory already holds under $\mathbf{k}_t$. If a program sets `rate = 0.05` and later `rate = 0.07`, the memory returns the sum of both values, scaled by the decays, when queried with the key of `rate`. A decay can make the old value fade, but it fades every other pair as well. The next section changes the write so that it replaces the value stored under a key.

12.4.5 Summary

A matrix-valued state stores key–value pairs: a write adds the outer product $\mathbf{v}_t\mathbf{k}_t^\top$, and a read multiplies the state with a query. Unrolled, this recurrence is causal linear attention, and the selective state space model of Section 12.3 is a memory of this kind with the step-scaled input as value and $\mathbf{B}_t$ and $\mathbf{C}_t$ as key and query. Without decay, every stored pair interferes with every read: for random unit keys the expected squared read error is $(n-1)/d_k$ after n writes. An input-dependent decay lets old pairs fade. With a scalar decay per head, the outputs equal masked attention with a learned, decaying mask, $(\mathbf{L}\odot\mathbf{Q}\mathbf{K}^\top)\mathbf{V}$, and computing this product in chunks, attention within a chunk and a recurrence over the chunk boundaries, trains the model with matrix products in memory linear in the length. A decay per key coordinate, as in gated linear attention, gives each coordinate a mask of its own; computing every decay over a stretch of time keeps all factors at most one, while folding the decays into queries and keys overflows. The layer surrounds the memory with a short convolution, unit queries and keys, and a normalized, gated output. Every write adds, so a key that is written twice returns the sum of its values.

12.4.6 Exercises

1. **Mamba as a matrix memory.** Write the selective state space model of Section 12.3 with its channel states as the rows of $\mathbf{S}_t \in \mathbb{R}^{H\times N}$, and drop the skip term.
   1. Write the update of $\mathbf{S}_t$ as an elementwise product of $\mathbf{S}_{t-1}$ with a decay matrix $\mathbf{E}_t$, plus an outer product, and the output as a matrix–vector product. Give every entry of $\mathbf{E}_t$.
   2. Under which condition on Δ_t and on the rates λ_n is $\mathbf{E}_t$ a scalar multiple of the matrix of ones, so that the update has the form Equation 12.25? Which of these conditions does Mamba-2 impose?
   3. For each form of decay in this section, count the input-dependent numbers per token that determine the decay of one head.
2. **Rank-one blocks.** Let $p_t = a_1 a_2 \cdots a_t$ be the products of the decays in Equation 12.26.
   1. Express L_{ts} through the p_t, and show that every submatrix of $\mathbf{L}$ whose row indices are all at least as large as its column indices has rank one.

   2. Deduce that every block of $\mathbf{L} \odot \mathbf{Q}\mathbf{K}^\top$ that couples a chunk to an earlier chunk has rank at most d_k. Identify where `chunked_linear_attention` applies the query-side and the key-side decays of such a block.
   3. With a decay per key coordinate, $\mathbf{S}_t = \mathbf{S}_{t-1}\mathrm{diag}(\alpha_t) + \mathbf{v}_t\mathbf{k}_t^\top$, no single mask $\mathbf{L}$ multiplies $\mathbf{Q}\mathbf{K}^\top$. Write the decayed score of query t and key s, and show that a block coupling two chunks still has rank at most d_k.
3. **Counting operations.** Count one multiply-add as two floating-point operations, set $d_k = d_v = d$, and keep the leading terms.
   1. Count the operations of the recurrence, the dual form, and the chunked form at length T and chunk size m. Split the chunked count into the three parts of the algorithm.
   2. Show that the chunked form costs at most a small constant times the recurrence while $m \le d$, and compute the ratio of chunked to dual operations at $T = 4096$, $d = 64$, and $m = 256$.
   3. Model the run time of a schedule as its operations divided by a throughput P, plus a fixed time τ per sequential step: T steps for the recurrence and T/m for the chunked form. Find the chunk size that minimizes the chunked time, and evaluate it for $T = 4096$, $d = 64$, $P = 10^{13}$ operations per second, and $\tau = 5\,\mu$s. Compare with the timings above.
4. code **The normalizer as a value column.** The linear attention of Section 10.5 applies a positive feature map ϕ to queries and keys and divides each output by $\phi(\mathbf{q}_t)^\top \mathbf{z}_t$, where $\mathbf{z}_t = \sum_{s \le t} \phi(\mathbf{k}_s)$. Give $\mathbf{z}_t$ the same decay as the state.
   1. Show that appending a constant 1 to every value vector makes the last coordinate of the output equal to the denominator.
   2. Compute the normalized outputs with one call of `chunked_linear_attention`, with $\phi = \mathrm{elu} + 1$, and check them against a loop that carries $\mathbf{S}_t$ and $\mathbf{z}_t$.
   3. Which of the two changes, the feature map or the normalizer, alters the relative weights that a query gives to past positions, and which only rescales each output?
5. **Targeted forgetting.** Let a memory hold $\mathbf{S} = \mathbf{v}\mathbf{k}^\top + \sum_i \mathbf{v}_i\mathbf{k}_i^\top$, where $\mathbf{k}$ has unit length and every $\mathbf{k}_i$ is orthogonal to $\mathbf{k}$.
   1. Find a matrix $\mathbf{D}$ such that $\mathbf{S}\mathbf{D} = \sum_i \mathbf{v}_i\mathbf{k}_i^\top$ for all such memories. Is it unique?
   2. For which keys $\mathbf{k}$ is this transition one of the decays of this section?
   3. Apply the transition and then write a new value, $\mathbf{S}\mathbf{D} + \mathbf{v}'\mathbf{k}^\top$. Show that the memory now returns $\mathbf{v}'$ for $\mathbf{k}$ and is otherwise unchanged. The next section builds a layer on this update.
6. **Correlated keys.** ▯ The keys with $\rho = 0.5$ in `matrix_memory` are normalized versions of $\sqrt{1-\rho^2}\,\mathbf{g}_i + \rho\,\mathbf{c}$, with independent random unit vectors $\mathbf{g}_i$ and one shared unit vector $\mathbf{c}$. Model the overlap of two distinct keys as $\mathbf{k}_i^\top \mathbf{k}_j = \rho^2 + \epsilon_{ij}$, where the ϵ_{ij} with $i \ne j$ are independent with mean zero and variance $(1-\rho^2)^2/d_k$, and neglect the normalization.
   1. Assume that the values are independent unit vectors with mean zero. Derive the expected squared read error at key j, which generalizes Equation 12.24, and identify the term that a larger d_k does not reduce.
   2. The values of `matrix_memory` come from a codebook of 256 random vectors, whose mean $\bar{\mathbf{v}}$ is not exactly zero. Show that this adds a term proportional to $(n-1)(n-2)\,\|\bar{\mathbf{v}}\|^2$, and compare your complete prediction with the measured errors.
7. code **Decay as a sliding window.** ▯ Add a constant decay a to `matrix_memory`, so that the pair written k steps before the end enters the final memory with weight a^k.
   1. At $d_k = 64$ and $n = 256$, measure recall separately for the oldest and the newest

quarter of the pairs, for $a \in \{1, 0.99, 0.95, 0.9\}$. What does the decay give the newest pairs, and what does it cost the oldest?

2. Following the proof of the proposition, show that the expected squared interference at any key is at most $1/((1-a^2)\,d_k)$. Show that the number of pairs whose squared signal a^{2k} exceeds this bound is at most about d_k/e, whatever $a < 1$ is.

8. code **Sub-chunks.** ⊠ The masks of `chunked_linear_attention` with a decay per key coordinate take $d_k m^2$ numbers per chunk.
   1. Compare their memory with that of a single mask, at $m = 64$ and $d_k = 64$.
   2. Cut each chunk into sub-chunks of 16 steps. For a query t and a key s in an earlier sub-chunk, let r be the last step before the sub-chunk of t, and write $\exp(c_t - c_s) = \exp(c_t - c_r)\exp(c_r - c_s)$. Show that both factors lie in $(0, 1]$, so that the scores between sub-chunks are a plain matrix product of scaled queries and keys that cannot overflow.
   3. Implement the masks within sub-chunks and the products between them, check the result against the loop for chunk sizes 64 and 256, and compare memory and run time with `chunked_linear_attention`. The kernels of Yang, Wang, Shen, et al. (2024) work this way.
9. code **One part at a time.** `PlainLinearAttention` drops three parts of gated linear attention at once. Remove the short convolution, the SiLU, and the output gate one at a time, and train each variant with `d2l.fit_lm`. Which part accounts for most of the gap in perplexity?

12.5 The Delta Rule

The matrix memories of Section 12.4 add each new pair to what they already hold, which works as long as each name is bound once. Text often binds a name again. A program sets `rate = 0.05` and later `rate = 0.07`, and from then on only the new value is correct. A story moves its hero from Paris to Rome; a dialogue corrects an earlier date. A memory that stores a pair by adding $\mathbf{v}_t\mathbf{k}_t^\top$ cannot follow such a change: a key written twice returns the sum of its two values, and the more often a key is rewritten, the less the latest value stands out. How can a memory replace what it has stored under a key?

The fix is to read before writing. Before storing a value under a key, the memory computes what it currently returns for that key and stores only the difference between the new value and that prediction. This *delta rule* is one step of gradient descent on the error of recalling the new value, and it turns the memory into a model that the network trains on the sequence it reads, one step per token. We derive the rule, give it a decay as well (Gated DeltaNet), train it in chunks like the layers of the previous section, and then look at all the memories of this chapter as solutions of one regression problem that differ in how they solve it. However the memory writes, its size stays fixed; the last section of the chapter asks what that means for recall.

12.5.1 Additive Writes

12.5.1.1 A Key Written Twice

Before changing the write, we measure how badly addition does when a key is rewritten. Start with a single key $\mathbf{k}$ of unit length. Write the value 1 under it, then the value 2: the memory is $\mathbf{S} = 1 \cdot \mathbf{k}^\top + 2 \cdot \mathbf{k}^\top$, and reading it returns $\mathbf{S}\mathbf{k} = 3$. In general, if a key is written R times with values $\mathbf{v}^{(1)}, \dots, \mathbf{v}^{(R)}$, the additive memory returns

$$\mathbf{S}\mathbf{k} = \mathbf{v}^{(1)} + \mathbf{v}^{(2)} + \cdots + \mathbf{v}^{(R)}, \tag{12.27}$$

plus the interference from other keys, $\sum_i (\mathbf{k}_i^\top \mathbf{k})\, \mathbf{v}_i$ over the pairs stored under other keys $\mathbf{k}_i$ (Section 12.4). The latest value carries the same weight as every stale one. A decay does not help much: it shrinks the stale values, but it shrinks the values of every other key by the same factor, whether or not they were rewritten.

To measure the effect apart from interference, we store values under 8 orthonormal keys in a memory of width 64, so that keys do not overlap at all. Each key is written R times, each time with a value drawn from a codebook of 32 random unit vectors, and we count how often the nearest codebook entry to the read is the latest value. The function takes the write rule as an argument, so that we can compare rules.

```
def overwrite_recall(write, num_writes, d=64, num_keys=8, num_values=32,
                     trials=200, seed=0):
    """Recall of the latest value after every key is written num_writes times."""
    rng = np.random.default_rng(seed)
    codebook = rng.standard_normal((num_values, d))
    codebook /= np.linalg.norm(codebook, axis=1, keepdims=True)
    recall = 0
    for _ in range(trials):
        keys = np.linalg.qr(rng.standard_normal((d, num_keys)))[0].T  # Rows
        S = np.zeros((d, d))
        for _ in range(num_writes):
            ids = rng.integers(0, num_values, num_keys)
            for k, v in zip(keys, codebook[ids]):
                S = write(S, k, v)
        reads = keys @ S.T  # Row j is S k_j
        recall += ((reads @ codebook.T).argmax(1) == ids).mean() / trials
    return recall

additive = lambda S, k, v: S + np.outer(v, k)
writes = [1, 2, 4, 8]
print('additive:', ' '.join(f'{overwrite_recall(additive, R):.2f}'
                            for R in writes))
```

```
additive: 1.00 0.51 0.27 0.21
```

After one write per key, every value is read back. After two writes, the latest value wins only about half of the time, which is what a sum of two equally weighted values allows; after eight, it wins about one time in five. The memory has room for all eight keys, and it still fails, because it cannot replace a value.

12.5.2 Writes as Gradient Steps

12.5.2.1 The Delta Rule

We want a write that makes the memory return $\mathbf{v}_t$ for $\mathbf{k}_t$ and changes as little else as possible. One way to get it is to measure how badly the memory recalls the new pair, with the squared error

$$\ell_t(\mathbf{S}) = \tfrac{1}{2} \left\| \mathbf{S}\mathbf{k}_t - \mathbf{v}_t \right\|^2, \tag{12.28}$$

and to take one step of gradient descent on it. The gradient is $\nabla \ell_t(\mathbf{S}) = (\mathbf{S}\mathbf{k}_t - \mathbf{v}_t)\mathbf{k}_t^\top$, the outer product of the *prediction error*, what the memory answers minus what it should answer, with the key. A step of size β_t from $\mathbf{S}_{t-1}$ gives

$$\mathbf{S}_t = \mathbf{S}_{t-1} - \beta_t(\mathbf{S}_{t-1}\mathbf{k}_t - \mathbf{v}_t)\mathbf{k}_t^\top. \tag{12.29}$$

This is the *delta rule*, named for the difference it writes, and it is old: Widrow and Hoff (1960) trained adaptive linear filters with it, and Schlag et al. (2021) proposed it as the write of a linear attention layer. Read it as two steps. First, read the memory at the key, $\hat{\mathbf{v}}_t = \mathbf{S}_{t-1}\mathbf{k}_t$. Then add the correction $\beta_t(\mathbf{v}_t - \hat{\mathbf{v}}_t)\mathbf{k}_t^\top$. If the memory already returns $\mathbf{v}_t$, nothing is written.

For a key of unit length and $\beta_t = 1$, reading right after the write returns $\mathbf{S}_t\mathbf{k}_t = \hat{\mathbf{v}}_t + (\mathbf{v}_t - \hat{\mathbf{v}}_t) = \mathbf{v}_t$: the old value is replaced. For a direction $\mathbf{u}$ orthogonal to $\mathbf{k}_t$, $\mathbf{S}_t\mathbf{u} = \mathbf{S}_{t-1}\mathbf{u}$: what is stored under other keys is untouched. For β_t between 0 and 1, the value stored under $\mathbf{k}_t$ moves to $(1-\beta_t)\hat{\mathbf{v}}_t + \beta_t\mathbf{v}_t$, a blend like the update gate of a GRU, so β_t is a *write strength*. The network computes it from the input with a sigmoid, and like the decay it is a gate.

The additive write of Section 12.4 is also a gradient step, on a different objective. Its update $\mathbf{v}_t\mathbf{k}_t^\top$ is the gradient of $\mathbf{v}_t^\top\mathbf{S}\mathbf{k}_t$, which rewards the memory for aligning its read with the value but never penalizes it for overshooting. Aligning gives superposition; minimizing the error gives correction. We repeat the measurement with the delta rule at $\beta = 1$.

```
delta = lambda S, k, v: S + np.outer(v - S @ k, k)
print('delta:   ', ' '.join(f'{overwrite_recall(delta, R):.2f}'
                            for R in writes))
```

```
delta:    1.00 1.00 1.00 1.00
```

The delta rule recalls the latest value every time, however often the key was rewritten.

12.5.2.2 The Transition

Expanding Equation 12.29 separates what happens to the old memory from what is written:

$$\mathbf{S}_t = \mathbf{S}_{t-1}(\mathbf{I} - \beta_t\mathbf{k}_t\mathbf{k}_t^\top) + \beta_t\mathbf{v}_t\mathbf{k}_t^\top. \tag{12.30}$$

The write is the additive write, scaled by β_t. The difference is the transition. In Section 12.4 the old memory was multiplied by a number a_t; here it is multiplied by the matrix $\mathbf{I} - \beta_t\mathbf{k}_t\mathbf{k}_t^\top$, which depends on the current key. For a unit key, this matrix leaves every direction orthogonal to $\mathbf{k}_t$ unchanged, eigenvalue 1, and scales the direction $\mathbf{k}_t$ by $1-\beta_t$ (Figure 12.9). At $\beta_t = 1$ it removes the $\mathbf{k}_t$ component of every row of the memory, which deletes the old value and makes room for the new one; for $\beta_t \in (0,1)$ it shrinks that component; at $\beta_t = 2$ it reverses it, a reflection. The decay of the previous section forgets in every direction at once. The delta rule forgets in one direction, the one it is about to write.

The eigenvalue $1 - \beta_t$ gives two guarantees. For $0 \le \beta_t \le 2$ and unit keys, no direction is ever amplified, so the memory cannot grow without bound, which is why delta-rule layers scale their keys to unit length. And for β_t above 1 the eigenvalue is negative, which no decay in $(0,1)$ can produce. A transition with a negative eigenvalue can flip the sign of a stored component and flip it back, and so track whether it has seen an even or an odd number of some token: $\beta_t = 2$ on that token and $\beta_t = 0$ elsewhere computes the parity of

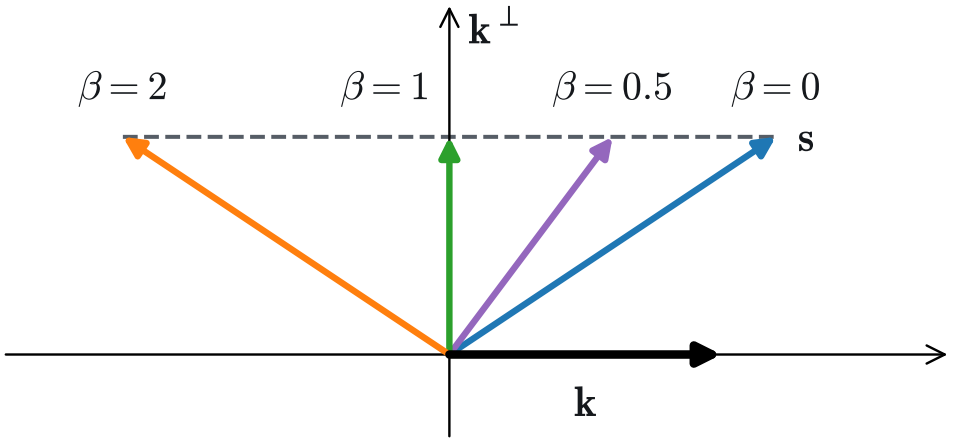


Figure 12.9: The transition $\mathbf{I} - \beta\mathbf{k}\mathbf{k}^\top$ acting on a vector $\mathbf{s}$ in the plane spanned by a unit key $\mathbf{k}$ and an orthogonal direction. The component along $\mathbf{k}$ is multiplied by $1-\beta$: kept at $\beta = 0$, halved at $\beta = 0.5$, removed at $\beta = 1$, and reversed at $\beta = 2$. The orthogonal component is unchanged.

a sequence of any length exactly. Grazzi et al. (2025) show that linear recurrences whose transitions have only eigenvalues between zero and one cannot compute parity at arbitrary lengths. This includes the decays of the previous sections and the delta rule with $\beta_t \leq 1$, and allowing β_t up to 2 removes the limitation in practice.

12.5.3 Gated DeltaNet

12.5.3.1 Decay and Correction

The delta rule edits only the direction it writes. Components of the memory along keys that the sequence never revisits are never cleared, and at the start of a new topic everything stored earlier stays. Decay and correction do different jobs, and Gated DeltaNet (S. Yang et al. 2025) combines them: the memory decays by an input-dependent a_t and then takes a delta-rule step,

$$\mathbf{S}_t = a_t\, \mathbf{S}_{t-1}(\mathbf{I} - \beta_t \mathbf{k}_t \mathbf{k}_t^\top) + \beta_t \mathbf{v}_t \mathbf{k}_t^\top . \tag{12.31}$$

Replacing $\mathbf{I} - \beta_t \mathbf{k}_t \mathbf{k}_t^\top$ by $\mathbf{I}$ gives the decayed memory of Section 12.4; with $a_t = 1$ the update is the delta rule. The model can clear its memory by setting a_t near zero and rebind one key by setting β_t near one.

12.5.3.2 Chunked Computation

The delta rule is a linear recurrence in $\mathbf{S}$, but its transition is a $d_k \times d_k$ matrix rather than a number, so the dual form of Section 12.4 does not apply directly, and a scan would compose dense matrices. Chunks still work, with one additional step. Write the update as $\mathbf{S}_t = a_t \mathbf{S}_{t-1} + \mathbf{u}_t \mathbf{k}_t^\top$, where $\mathbf{u}_t = \beta_t(\mathbf{v}_t - a_t \mathbf{S}_{t-1}\mathbf{k}_t)$ is the corrected value that is actually written. If we knew the $\mathbf{u}_t$, the memory would be the decayed additive memory of the previous section

with values $\mathbf{u}_t$, and the chunked algorithm there would compute its outputs. The $\mathbf{u}_t$ depend on each other, but only through the overlaps of the keys. Within a chunk that starts with memory $\mathbf{S}_0$, unrolling $\mathbf{S}_{t-1}$ gives

$$\mathbf{u}_t + \beta_t \sum_{s<t} L_{ts} \, (\mathbf{k}_s^\top \mathbf{k}_t) \, \mathbf{u}_s = \beta_t (\mathbf{v}_t - \gamma_t \mathbf{S}_0 \mathbf{k}_t),$$

where L_{ts} is the product of the decays from step $s+1$ to t, as in the dual form of Section 12.4, and γ_t is the product of the decays since the chunk started. Stacked as rows, these are m linear equations for the $\mathbf{u}_t$ with a lower triangular matrix: each $\mathbf{u}_t$ depends only on earlier ones. One triangular solve per chunk replaces m sequential steps, and because the right side is linear in $\mathbf{S}_0$, we solve once for the part that depends on the values and once for the part that depends on the incoming memory, before the state is known. The *WY representation*, a classical way of writing products of Householder-like matrices, organizes the same computation; Yang, Wang, Zhang, et al. (2024) adapted it to the delta rule.

We split the computation into two functions. The first solves, for every chunk at once, the triangular systems for the two parts of the corrected values: $\mathbf{W}$, which depends on the values, and $\mathbf{Z}$, which multiplies the incoming memory, so that $\mathbf{U} = \mathbf{W} - \mathbf{Z}\mathbf{S}_0^\top$. It uses the mask `decay_mask` of Section 12.4, whose entries are the products L_{ts}.

```
def delta_rule_writes(K, V, beta, log_a):
    """W and Z with corrected values U = W - Z S_0^T, for every chunk.

    K: (..., m, d_k); V: (..., m, d_v); beta, log_a: (..., m)
    """
    L = d2l.decay_mask(log_a)  # Products of decays within the chunk
    gamma = torch.exp(log_a.cumsum(-1)).unsqueeze(-1)  # Decay since start
    beta = beta.unsqueeze(-1)
    M = torch.eye(K.shape[-2], device=K.device) + (
        beta * L.tril(-1) * (K @ K.mT))  # Lower triangular
    W = torch.linalg.solve_triangular(M, beta * V, upper=False)
    Z = torch.linalg.solve_triangular(M, beta * gamma * K, upper=False)
    return W, Z
```

The second passes the memory from chunk to chunk. Because the corrected values $\mathbf{U} = \mathbf{W} - \mathbf{Z}\mathbf{S}_0^\top$ are affine in the incoming memory, so is the memory at the chunk's end: $\mathbf{S}_0\mathbf{A} + \mathbf{W}_{\text{end}}$, where the $d_k \times d_k$ matrix $\mathbf{A}$ is the decay of the chunk minus what its corrections remove, and $\mathbf{W}_{\text{end}}$ collects the chunk's own writes. Affine maps compose like the steps of the linear recurrences of Section 12.2, with matrix products in place of elementwise ones, so a scan over the chunks gives the memory at the start of every chunk. From these, the corrected values and the outputs of all chunks follow at once, as in the chunked computation of Section 12.4 with the values $\mathbf{U}$ in place of $\mathbf{V}$.

```
def chunked_delta_rule(Q, K, V, beta, log_a, chunk_size):
    """Outputs y_t = S_t q_t of the gated delta rule, computed in chunks.

    Q, K: (batch_size, T, d_k); V: (batch_size, T, d_v);
    beta, log_a: (batch_size, T)
    """
```

```
    B, T, m = Q.shape[0], Q.shape[1], chunk_size
    Q, K, V = (X.reshape(B, T // m, m, -1) for X in (Q, K, V))
    beta, log_a = beta.reshape(B, T // m, m), log_a.reshape(B, T // m, m)
    W, Z = delta_rule_writes(K, V, beta, log_a)
    L, cum = d2l.decay_mask(log_a), log_a.cumsum(-1)
    # Each chunk maps the memory at its start to S A + W_end at its end
    to_end = torch.exp(cum[..., -1:] - cum).unsqueeze(-1)
    I = torch.eye(K.shape[-1], device=K.device)
    A = torch.exp(cum[..., -1, None, None]) * I - (Z * to_end).mT @ K
    def combine(earlier, later):  # A scan over chunks, with matrix products
        (A1, W1), (A2, W2) = earlier, later
        return A1 @ A2, W1 @ A2 + W2
    W_end = d2l.associative_scan(combine, (A, (W * to_end).mT @ K))[1]
    S = torch.cat([torch.zeros_like(W_end[:, :1]), W_end[:, :-1]], 1)
    U = W - Z @ S.mT  # Corrected values, with the memory at each start
    Y = torch.exp(cum).unsqueeze(-1) * (Q @ S.mT) + (L * (Q @ K.mT)) @ U
    return Y.reshape(B, T, -1)
```

We check the chunked computation against the recurrence Equation 12.31 applied one step at a time, with unit keys and random write strengths and decays.

```
def delta_rule_recurrent(Q, K, V, beta, a):
    """The same outputs, one step at a time."""
    S, outputs = Q.new_zeros(V.shape[-1], K.shape[-1]), []
    for q, k, v, b, a_t in zip(Q, K, V, beta, a):
        S = a_t * S
        S = S + b * torch.outer(v - S @ k, k)
        outputs.append(S @ q)
    return torch.stack(outputs)
```

```
gen = torch.Generator().manual_seed(0)  # Local: later training stays unseeded
T, d_k, d_v = 256, 32, 32
Q, V = torch.randn(T, d_k, generator=gen), torch.randn(T, d_v, generator=gen)
K = F.normalize(torch.randn(T, d_k, generator=gen), dim=-1)
beta = torch.rand(T, generator=gen)
a = 0.8 + 0.19 * torch.rand(T, generator=gen)
Y = delta_rule_recurrent(Q, K, V, beta, a)
for m in [16, 64]:
    Y_chunked = chunked_delta_rule(Q[None], K[None], V[None], beta[None],
                                   torch.log(a)[None], m)[0]
    print(f'chunk size {m}: relative deviation '
          f'{(Y_chunked - Y).abs().max() / Y.abs().max():.1e}')
```

```
chunk size 16: relative deviation 2.1e-07
chunk size 64: relative deviation 3.1e-07
```

The chunked outputs agree with the recurrence to rounding error.

12.5.3.3 The Layer

A Gated DeltaNet layer is the gated linear attention layer of Section 12.4 with a different memory. It adds one projection, which computes a write strength $\beta_t = \sigma(\mathbf{w}^\top \mathbf{x}_t + b)$ for

every head, and replaces the method `memory` by the chunked delta rule. Its decay is one number per head, $a_t = \sigma(\mathbf{w}_a^\top \mathbf{x}_t + b_a)$, as in S. Yang et al. (2025), because the chunked delta rule above takes one decay per step; the method `decays` changes accordingly, and its biases start the heads with memories of 4 to 256 steps. The short convolution, the unit queries and keys, and the output norm and gate stay as they were. Unit keys matter more here than before: without them, the eigenvalue $1 - \beta_t \|\mathbf{k}_t\|^2$ of the transition can fall below -1, and the memory can grow without bound.

```
class GatedDeltaNet(d2l.GatedLinearAttention):
    """Gated linear attention with one decay per head, whose memory is
    written by the delta rule."""
    gate_scale = 1  # For init_decays: log sigmoid(b) = -1 / tau

    def __init__(self, num_hiddens, num_heads, chunk_size=32, conv_width=4,
                 timescales=(4, 256)):
        super().__init__(num_hiddens, num_heads, chunk_size, conv_width,
                         timescales)
        self.W_beta = nn.Linear(num_hiddens, num_heads)  # Write strength
        d2l.init_lecun(self.W_beta)

    def decay_gate(self, d):  # One gate per head
        # Sequential, so that init_decays finds the bias at W_a[-1]
        return nn.Sequential(nn.Linear(d, self.num_heads))

    def decays(self, X):  # (batch_size * num_heads, T)
        return d2l.heads_to_batch(F.logsigmoid(self.W_a(X)),
                                  self.num_heads).squeeze(-1)

    def memory(self, X, Q, K, V, log_a):
        beta = d2l.heads_to_batch(torch.sigmoid(self.W_beta(X)),
                                  self.num_heads).squeeze(-1)
        return chunked_delta_rule(Q, K, V, beta, log_a, self.chunk_size)
```

We train it as the mixer of `d2l.SeqLM` with the recipe of the chapter: character-level Penn Treebank, two blocks of width 128 with four heads, and `d2l.fit_lm`. The gated linear attention layer reached a perplexity of about 3.1 with the same recipe.

```
data = d2l.PennTreebank(batch_size=64, num_steps=128)
```

```
model = d2l.SeqLM(len(data.vocab), 128,
                  [GatedDeltaNet(128, num_heads=4) for _ in range(2)])
print(f'Gated DeltaNet perplexity: {d2l.fit_lm(model, data):.2f}')
```

```
Gated DeltaNet perplexity: 3.13
```

Gated DeltaNet reaches a perplexity of about 3.1, the same as gated linear attention. On this task the write rule matters little: most predictions depend on the last few characters, and few keys need to be rebound within a window of 128 characters. The difference shows where a task requires it, in the overwrite measurement above and in the recall experiments of the next section.

12.5.4 Memory as Regression

Every memory in this chapter answers a query with a function fitted to the pairs $(\mathbf{k}_s, \mathbf{v}_s)$ it has read. Asking which function, and how it is fitted, puts the models side by side. Collect the keys and values of the first t steps as the rows of $\mathbf{K}_t$ and $\mathbf{V}_t$. The linear map that reproduces the stored pairs best, in the sense of least squares, minimizes $\sum_{s \le t} \|\mathbf{S}\mathbf{k}_s - \mathbf{v}_s\|^2$. When $\mathbf{K}_t^\top \mathbf{K}_t$ is invertible, the minimizer is

$$\mathbf{S}_t^\star = \mathbf{V}_t^\top \mathbf{K}_t \, (\mathbf{K}_t^\top \mathbf{K}_t)^{-1}. \tag{12.32}$$

The first factor, $\mathbf{V}_t^\top \mathbf{K}_t = \sum_s \mathbf{v}_s \mathbf{k}_s^\top$, is the additive memory of Section 12.4. The second factor undoes the overlaps of the keys. For orthonormal keys it is the identity on the directions they span, and the additive memory is already the best fit there; for overlapping keys, it removes the interference between them, the terms $(\mathbf{k}_i^\top \mathbf{k}_j)\, \mathbf{v}_i$ that an additive memory adds to every read. Linear attention is therefore least squares without the correction for overlapping keys. The delta rule is a gradient step on the same objective, one pair at a time, and a key that is written again is fitted again. With a decay, linear attention weights the pairs so that recent ones count more, while in Gated DeltaNet the decay shrinks the fitted memory toward zero before each gradient step, a weight decay. Softmax attention fits nothing in advance. It keeps every pair and, at query time, returns a weighted average of the values with weights given by a kernel of the query and the keys, the Nadaraya–Watson estimator of Section 10.1.2. This is a nonparametric regression, which is why its memory grows with the context. Table 12.1 collects the view; K. A. Wang et al. (2025) develop it in detail.

Table 12.1: Memories as regressions on the pairs read so far.

Memory	What it fits	How	State
Softmax attention	kernel-weighted average of all values	at query time	all pairs
Linear attention	$\mathbf{V}^\top \mathbf{K}$, least squares without the key correction	one addition per pair	$d_v \times d_k$
Gated linear attention	the same, weighted toward recent pairs	one addition per pair	$d_v \times d_k$
Delta rule	least squares	one gradient step per pair	$d_v \times d_k$
Gated DeltaNet	least squares with weight decay	one gradient step per pair	$d_v \times d_k$

The view suggests how to design new memories: choose the weights of the pairs, the class of functions the memory fits, and the method that fits it. As an example of a different method, consider replacing the gradient step by the *implicit* step that minimizes $\frac{1}{2}\|\mathbf{S} - \mathbf{S}_{t-1}\|^2 + \frac{\beta_t}{2}\|\mathbf{S}\mathbf{k}_t - \mathbf{v}_t\|^2$ exactly: the new memory should recall the new pair well and stay close to the old memory. Its solution is the delta rule with the step $\beta_t/(1 + \beta_t\|\mathbf{k}_t\|^2)$ in place of β_t, as an exercise shows, so the eigenvalue along $\mathbf{k}_t$ becomes $1/(1 + \beta_t\|\mathbf{k}_t\|^2)$, which lies in $(0, 1)$ for every positive β_t and every key. The implicit step cannot overshoot, whatever the network sets the write strength to; this is the update of Longhorn (B. Liu et al. 2024). Other

designs replace the matrix by a small neural network that is trained by gradient steps as the sequence is read (Y. Sun et al. 2024; Behrouz et al. 2025).

In all of these models, learning happens at two levels. While the model reads a sequence, its memory is fitted to that sequence's pairs, at training time and when the model is used, and it starts empty for every new sequence. The network's weights, which compute the keys, values, queries, and gates from the input, are trained by backpropagation through these fitting steps, over many sequences, and stay fixed when the model is used. The weights learn which pairs are worth storing; the memory stores the pairs of the sequence at hand.

Whatever its write rule, the memory has $d_v \times d_k$ numbers, and a fixed set of numbers can hold only so many facts. The last section of the chapter measures this limit on a recall task and shows how a few attention layers remove it.

12.5.5 Summary

An additive matrix memory cannot rebind a key: a key written several times returns the sum of its values, and recall of the latest value falls to about one half after two writes. The delta rule reads the memory before it writes and stores only the difference between the new value and the current prediction. This is one gradient step on the squared error of recalling the new pair, and in transition form it multiplies the memory by $\mathbf{I} - \beta_t \mathbf{k}_t \mathbf{k}_t^\top$, which removes or shrinks only the direction being written. Gated DeltaNet adds a decay, so that the memory can both clear itself and correct a single binding. The chunked algorithm of the previous section extends to the delta rule with one triangular solve per chunk. Viewed as regressions, linear attention keeps the cross-moment of keys and values, the delta rule fits least squares by gradient steps, and softmax attention keeps all pairs and smooths among them at query time.

12.5.6 Exercises

1. **Order does not matter.** Show that the additive memory after a sequence of writes does not depend on the order of the writes. Conclude that no read rule, however clever, can recover the latest of R values written under one key from an additive memory alone, and compute the recall that the best rule achieves when the R values are distinct. Compare with the measured recall.

2. code **Decay instead of correction.** Give the additive write of `overwrite_recall` a constant decay, $\mathbf{S} \leftarrow a\mathbf{S} + \mathbf{v}\mathbf{k}^\top$.

    1. For $R = 2, 4, 8$, find the decay a that maximizes the recall of the latest value. Explain the optimum: what does a decay close to zero cost?
    2. Now store, in addition, 8 keys that are written once at the start and never rewritten, and measure their recall at the end along with that of the rewritten keys. Which write rule serves both kinds of key?

3. **Step size and key length.** Let the keys of the delta rule have any length.

    1. Show that the eigenvalue of $\mathbf{I} - \beta\mathbf{k}\mathbf{k}^\top$ along $\mathbf{k}$ is $1 - \beta\|\mathbf{k}\|^2$, and find the range of β for which no direction is amplified. Relate the result to the step size limit $\eta < 2/L$ of gradient descent on a quadratic with curvature L.
    2. Show that dividing the step by $\|\mathbf{k}\|^2$, the *normalized* least mean squares rule, makes the update independent of the scale of the key. Why do delta-rule layers normalize the keys instead?

4. **The implicit step.** Show that the minimizer of $\frac{1}{2}\|\mathbf{S} - \mathbf{S}_{t-1}\|^2 + \frac{\beta}{2}\|\mathbf{S}\mathbf{k} - \mathbf{v}\|^2$ is $\mathbf{S}_{t-1} - \frac{\beta}{1+\beta\|\mathbf{k}\|^2}(\mathbf{S}_{t-1}\mathbf{k} - \mathbf{v})\mathbf{k}^\top$. (Hint: set the gradient to zero and multiply the result by $\mathbf{k}$ to find $\mathbf{S}\mathbf{k}$ first.) Give the eigenvalues of its transition, and show that the step recalls the new pair exactly only in the limit $\beta \to \infty$.

5. code **Correlated keys.** Use the correlated keys of the experiment in Section 12.4, with $\rho = 0.5$ and $d_k = 64$, and store n pairs once each.

    1. Compare the squared read error of the additive memory, of the delta rule with $\beta = 1$ applied once to each pair, and of the least squares solution Equation 12.32, for n from 8 to 64.
    2. Apply the delta rule for several passes over the same pairs. How many passes bring it close to least squares, and why does a single pass not suffice?

6. code **Recursive least squares.** The least squares solution Equation 12.32 can be updated one pair at a time if the memory also keeps $\mathbf{P}_t = (\mathbf{K}_t^\top \mathbf{K}_t + \lambda \mathbf{I})^{-1}$.

    1. Use the Sherman–Morrison formula to derive the updates of $\mathbf{P}_t$ and $\mathbf{S}_t$ after one new pair, and show that the update of $\mathbf{S}_t$ is a delta rule whose key is $\mathbf{P}_t\mathbf{k}_t$ instead of $\mathbf{k}_t$.
    2. Implement it and compare it with the delta rule on the overwrite experiment. Why does recursive least squares, the best fit of all pairs, recall the latest value poorly? What would fix it?
    3. How large is its state compared with that of the delta rule?

7. code **Parity.** ⊠ The write strength of `GatedDeltaNet` is a sigmoid, so its eigenvalues $1-\beta_t$ lie in $(0, 1)$.

    1. Build a one-head memory with keys fixed to a unit vector and write strengths set by hand from the input bits, and show that it computes the parity of bit strings of any length exactly. Which values of β_t does it need?
    2. Train a one-layer `SeqLM` with a `GatedDeltaNet` mixer to output the running parity of random bit strings of length 64. Then replace the sigmoid of the write strength by 2σ and train again. Compare the accuracies, and test both models on strings of length 256.

8. **Gradients through the memory.** The network's weights are trained by backpropagation through the writes. Consider a single delta-rule step with $\beta = 1$ from $\mathbf{S}_0 = \mathbf{0}$ followed by a read, $\mathbf{o} = \mathbf{S}_1\mathbf{q}$, and a loss $\frac{1}{2}\|\mathbf{o} - \mathbf{y}\|^2$.

    1. Compute the gradients of the loss with respect to $\mathbf{q}$, $\mathbf{k}$, and $\mathbf{v}$.
    2. Repeat for two writes with keys $\mathbf{k}_1$, $\mathbf{k}_2$ followed by a read with $\mathbf{q} = \mathbf{k}_1$. How does the gradient with respect to $\mathbf{k}_1$ depend on the overlap $\mathbf{k}_1^\top\mathbf{k}_2$? What does it encourage the network to do with the keys of different facts?

9. code **The parts around the memory.** Write a subclass of `GatedDeltaNet` without the short convolution, the SiLU, and the output gate, as `PlainLinearAttention` of Section 12.4 does for gated linear attention, and train it with `d2l.fit_lm`. Does the delta rule lose as much without these parts as the additive write does?

12.6 Hybrid Architectures

Every model in this chapter improved how a fixed-size state is written, from gates to selective dynamics to the delta rule, but none of them can make the state hold more than its size allows. Does that size limit what a model can recall from its context? Consider a user who pastes a table of 64 product codes and prices into a prompt and asks for the prices of 20 of them. A transformer answers such a request by looking back: every earlier token is still in its cache, and attention finds the ones that match. A model with a fixed-size state has only what it chose to store when the tokens went by, before it knew which of them would be asked for. Retrieval from the context is a large part of how language models are used, in few-shot prompts, in answers grounded in retrieved documents, and in editing code, where a programmer may ask for a 2,000-token function to be copied with one line changed. This limit therefore matters in practice.

This section measures the limit and shows the remedy that most recent large recurrent models adopt. We first count how much a fixed state can hold. We then train four small models on a recall task with a growing number of pairs: two built from the recurrent layers of the previous sections, one built from attention, and a *hybrid* in which one of its two layers is attention. We ask whether this one layer is enough to recall as well as a transformer, and what the hybrid saves in memory, by counting that memory for models of realistic size.

12.6.1 Limits of a Fixed-Size State

12.6.1.1 Counting Bits

The simplest limit comes from counting: a state cannot tell apart more inputs than it has configurations. A state of N numbers stored with b bits each can be in at most 2^{Nb} different configurations. Suppose a model reads k tokens, each drawn from a vocabulary of V tokens, and must later reproduce all of them. The V^k possible inputs must then lead to V^k different states, since two inputs that lead to the same state receive the same answer. This requires

$$Nb \geq k \log_2 V. \tag{12.33}$$

The state must hold $\log_2 V$ bits per token to be recalled, and the number of tokens it can recall grows in proportion to its size and not at all with the length of the context. A transformer's cache grows by one key and one value per token, so it never runs out. Jelassi et al. (2024) turn this argument into a lower bound on the state size needed to copy, and show that small transformers learn to copy strings far longer than recurrent models of the same size can.

The bound counts bits that a perfect encoding would use, and it is far from tight for the models below. Their two recurrent layers hold 128 numbers each, eight heads with a state of 4×4, so 256 numbers of 32 bits in all, many times the 128 bits that 64 values of 2 bits take. A matrix memory cannot pack bits like that: it writes by adding outer products and reads with a linear map, and the pairs it holds interfere. Section 12.4 showed that after n writes with random unit keys of dimension d_k, the expected squared read error is $(n-1)/d_k$. A trained model chooses its keys and values, however, and it can do much better than random ones.

12.6.1.2 Slots and Overlapping Keys

How much better? Suppose that a value needs one number, as a value from a small set does. A head with a $d_v \times d_k$ state can then give each key a slot of its own: the key's value goes into one row of the state, along a key direction orthogonal to those of the other keys assigned to that row. Reading with the key returns its value and nothing else. A row has room for d_k orthogonal directions, so a state of N numbers has N slots, and a model whose task draws keys from at most N possible ones can store any pairs without interference. With more possible keys than slots, some row must host more keys than it has dimensions, and these keys overlap, whatever training does. The next proposition says by how much.

Proposition. Let $\mathbf{k}_1, \dots, \mathbf{k}_m$ be unit vectors in $\mathbb{R}^d$, with $m > d$. Then the mean of $(\mathbf{k}_i^\top \mathbf{k}_j)^2$ over all pairs $i \neq j$ is at least

$$\frac{m-d}{d\,(m-1)}. \tag{12.34}$$

Proof. Let $\mathbf{G} = \sum_{i=1}^{m} \mathbf{k}_i \mathbf{k}_i^\top$. Its trace is $\sum_i \|\mathbf{k}_i\|^2 = m$, so its d eigenvalues sum to m, and by the Cauchy–Schwarz inequality their squares sum to at least m^2/d. The sum of the squared eigenvalues of the symmetric matrix $\mathbf{G}$ is $\|\mathbf{G}\|_F^2 = \sum_{i,j} (\mathbf{k}_i^\top \mathbf{k}_j)^2$, and the m terms with $i = j$ equal one. The other $m(m-1)$ terms therefore sum to at least $m^2/d - m$. ■

This is a bound of Welch (1974) . For $m \gg d$ it approaches $1/d$, the mean squared overlap of random unit keys in the proof of Equation 12.24. When a row hosts $m > d$ possible keys, a read of one of them picks up the values of the other stored keys of that row, as in Equation 12.23, with weights whose mean square is at least the bound. The more pairs a sequence stores, the more stored keys share a row, and the larger the interference.

The task below draws its keys from 512 possible ones, and the recurrent models have 256 numbers of state, so on average two possible keys share each slot. With a few pairs, the stored keys rarely meet in a row; with 64, many of them do. We therefore expect the recurrent models to recall nearly every pair when there are few, and to lose more and more of them as the number of pairs grows, since no training removes the overlap. Attention has no such limit, because its cache holds every pair it has read.

12.6.1.3 Associative Recall

To measure recall we use *multi-query associative recall* (Arora, Eyuboglu, Timalsina, et al. 2024). A sequence begins with n pairs, each a key token followed by a value token, for distinct keys drawn from 512 and values drawn from 4. Filler tokens follow, and the sequence ends with every key once more, in random order, as a query. At each query, the model must output the value that followed that key. Sequences have 256 tokens, so they hold up to 64 pairs with their queries. Arora, Eyuboglu, Timalsina, et al. (2024) found that this kind of recall accounts for most of the gap in perplexity on natural text between attention and the gated convolution models they studied. The 512 keys exceed the 256 slots of the recurrent models; the 4 values let one number hold a value.

```
def recall_batch(rng, batch_size, num_pairs, num_keys=512, num_values=4,
                 num_steps=256):
    """Pairs as a key token followed by a value token, then every key again
```

```
        as a query, in random order. Token ids: values, then keys, then filler."""
        keys = np.argsort(rng.random((batch_size, num_keys)), 1)[:, :num_pairs]
        values = rng.integers(0, num_values, (batch_size, num_pairs))
        order = np.argsort(rng.random((batch_size, num_pairs)), 1)
        X = np.full((batch_size, num_steps), num_values + num_keys)  # Filler
        X[:, 0:2 * num_pairs:2] = num_values + keys
        X[:, 1:2 * num_pairs:2] = values
        X[:, -num_pairs:] = num_values + np.take_along_axis(keys, order, 1)
        return X, np.take_along_axis(values, order, 1)  # Answers to the queries

    num_keys, num_values = 512, 4  # The defaults of recall_batch
    vocab_size = num_values + num_keys + 1  # And the filler
    X, Y = recall_batch(np.random.default_rng(), 1, 4)
    print('tokens:', X[0, :8], '...', X[0, -4:], ' answers:', Y[0])
```

```
tokens: [315   3 123   0 235   0   4   3] ... [235 315 123   4]  answers: [0 3 0 3]
```

12.6.2 Recall with One Attention Layer

12.6.2.1 Four Models

To see whether a single attention layer gives a recurrent model the recall of a transformer, we compare a hybrid with models built from one kind of layer. All four models are instances of `d2l.SeqLM` with two blocks of width 32; only the mixers differ. Two use the recurrent layers of the previous sections, gated linear attention (Section 12.4) and Gated DeltaNet (Section 12.5), with eight heads, so that keys and values have 4 dimensions. The third uses causal softmax attention in both blocks, and the fourth, the hybrid, replaces the second of its two Gated DeltaNet layers by attention. The attention mixer is computed like the multi-head attention of Chapter 10, with the fused attention function of each framework, behind the same short convolution as the recurrent layers, so that the mixers differ only in their memory. The attention mixer has no position encoding of its own, like the NoPE layers of Section 10.4, and the convolution gives it all the order it needs here: it brings each key into the position of the value that follows it, and a query then finds that position by content. Without it, two layers of attention with rotary position embeddings (Section 10.4.3) must form an *induction head*, the first copying each key into the position after it and the second looking the query up there. They learn this too, but only after a plateau of a few thousand training steps whose length varies from run to run, and some runs are still on it at the end of training. Attention gets two heads of dimension 16: its cache grows with the context and has no slots to count, so the size of its heads does not change the comparison.

```
class ConvAttention(nn.Module):
    """Causal multi-head softmax attention after a short convolution, with
    the interface of a mixer."""
    def __init__(self, num_hiddens, num_heads, conv_width=4):
        super().__init__()
        d = num_hiddens
        self.num_heads = num_heads
        self.W_qkv = nn.Linear(d, 3 * d, bias=False)
        self.conv = nn.Conv1d(3 * d, 3 * d, conv_width, groups=3 * d,
                              padding=conv_width - 1)  # One filter per channel
```

```
        self.W_o = nn.Linear(d, d, bias=False)
        d2l.init_lecun(self)

    def forward(self, X):  # X: (batch_size, num_steps, num_hiddens)
        B, T, d = X.shape
        QKV = self.conv(self.W_qkv(X).mT)[..., :T].mT  # Causal: drop the tail
        Q, K, V = (Z.reshape(B, T, self.num_heads, -1).transpose(1, 2)
                   for Z in QKV.chunk(3, -1))
        Y = F.scaled_dot_product_attention(Q, K, V, is_causal=True)
        return self.W_o(Y.transpose(1, 2).reshape(B, T, d))
```

A string describes the mixers of a model, one letter per block. The pairs must survive up to 256 steps, so the decay gates of the recurrent layers start with memories of 64 to 1,024 steps rather than the 4 to 256 steps of Section 12.4. As the long-range recall experiment of Section 12.2 showed, memories that start short are slow to learn to last: with the shorter start, the recurrent models below learn the task more slowly, gated linear attention markedly so.

```
def recall_model(layout, vocab_size, num_hiddens=32):
    """SeqLM whose mixers are given by letters: L(inear attention),
    G(ated DeltaNet), A(ttention)."""
    timescales = (64, 1024)  # Memories of 64 to 1,024 steps
    make = {'L': lambda: d2l.GatedLinearAttention(
                num_hiddens, num_heads=8, timescales=timescales),
            'G': lambda: d2l.GatedDeltaNet(
                num_hiddens, num_heads=8, timescales=timescales),
            'A': lambda: ConvAttention(num_hiddens, num_heads=2)}
    return d2l.SeqLM(vocab_size, num_hiddens, [make[c]() for c in layout])
```

12.6.2.2 The Recall Sweep

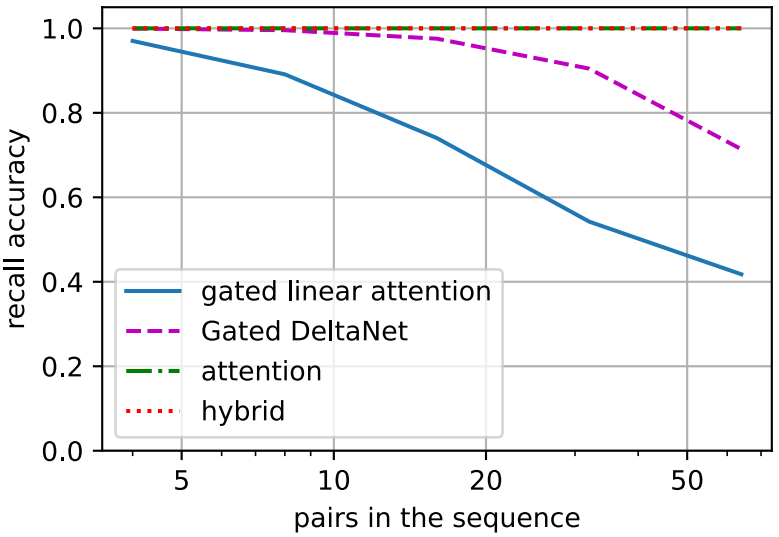
Each model is trained once on fresh batches whose number of pairs cycles through 4, 8, 16, 32, and 64, with the loss computed at the query positions only by `d2l.train_queries` of Section 12.1.4.1, and is then tested separately at each number of pairs. Training uses AdamW for 16,000 steps of 256 sequences, with gradients clipped to norm 1 and the learning rate of Section 7.3 (`d2l.cosine_warmup`): a linear warmup over 800 steps to 0.006, then a half-cosine down to zero.

```
def train_recall(model, pairs=(4, 8, 16, 32, 64), num_steps=16000,
                 batch_size=256, lr=6e-3, warmup=800):
    """Train with a cycling number of pairs; return accuracy per number."""
    rng = np.random.default_rng()
    model.to(d2l.try_gpu())
    optimizer = torch.optim.AdamW(model.parameters(), lr=lr, weight_decay=0.01)
    scheduler = torch.optim.lr_scheduler.LambdaLR(
        optimizer, lambda t: d2l.cosine_warmup(t, num_steps, warmup))
    batch = lambda step: recall_batch(rng, batch_size,
                                      pairs[step % len(pairs)])
    d2l.train_queries(model, batch, num_steps, optimizer, scheduler)
    return {n: float(d2l.query_accuracy(model, *recall_batch(rng, 512, n)))
            for n in pairs}
```

```
layouts = {'gated linear attention': 'LL', 'Gated DeltaNet': 'GG',
           'attention': 'AA', 'hybrid': 'GA'}
accuracy = {name: train_recall(recall_model(layout, vocab_size))
            for name, layout in layouts.items()}
```

```
pairs = list(next(iter(accuracy.values())))
print(f'{"pairs":24s}', ' '.join(f'{n:4d}' for n in pairs))
for name, acc in accuracy.items():
    print(f'{name:24s}', ' '.join(f'{acc[n]:.2f}' for n in pairs))
d2l.plot(pairs, [list(acc.values()) for acc in accuracy.values()],
         'pairs in the sequence', 'recall accuracy', legend=list(accuracy),
         xscale='log', ylim=(0, 1.05), figsize=(4.5, 3))
```

```
pairs                       4    8   16   32   64
gated linear attention   0.97 0.89 0.74 0.54 0.42
Gated DeltaNet           1.00 1.00 0.98 0.90 0.71
attention                1.00 1.00 1.00 1.00 1.00
hybrid                   1.00 1.00 1.00 1.00 1.00
```



With 4 pairs, all four models recall nearly every value. Beyond that Gated DeltaNet loses pairs, and the more pairs a sequence holds, the more it loses: it recalls about nine tenths of 32 pairs and about three quarters of 64. Gated linear attention recalls fewer from 8 pairs on, and its curve varies more from run to run. It first sits on a plateau, recalling about half of 4 pairs, for a number of training steps that varies from a few thousand to more than ten thousand, and only then learns the task. The reference implementation of the layer, from the flash-linear-attention library of its authors (Yang, Wang, Shen, et al. 2024), trained on the same task with the same recipe, sits on the same plateau for as long, in one of three runs for all 16,000 steps, so the plateau belongs to the model and not to our code. The hybrid recalls every pair at every load, although one of its two layers is recurrent, and so does attention in almost every run. The attention layer of the hybrid looks up the pairs that the recurrent state cannot hold.

The recurrent curves still rise slowly at the end of training. Trained twice as long, Gated DeltaNet recalls about 95% of 32 pairs and 80 to 90% of 64, and gated linear attention still recalls fewer. With keys drawn from 128 possible ones, fewer than the slots, the same long training brings Gated DeltaNet to 90% or more of 64 pairs; the gap to the task with 512 keys is the part of the curve that the slots explain. The delta rule pays off as the regression view of Section 12.5 suggests. An additive memory returns, with every value it reads, the values of all stored keys that overlap the query, while the corrected writes fit the stored pairs and remove much of that interference. Keys that share a slot overlap, so the correction matters most when the slots fill up.

The parts around the memory matter as much as the write rule. The value follows its key one token later, and the short convolution is what brings the two together for a single write. Without it, gated linear attention recalls barely more than chance even with 4 pairs, which fit easily into its state, and Gated DeltaNet learns more slowly and recalls fewer pairs at every load. Without unit keys, Gated DeltaNet does not learn the task at all. Without the output gate, Gated DeltaNet recalls several points less at 16 pairs and more; for gated linear attention the change stays within its spread from run to run. The SiLU after the convolution is the exception: it makes no difference on text, and on this task it costs Gated DeltaNet a few points at the highest loads.

12.6.2.3 Language Modeling

Does the difference show on ordinary text? We train two more models on the character-level Penn Treebank with the recipe of Section 12.1: the transformer `d2l.TinyLM` of Section 9.5, two blocks of width 128 with learned position embeddings, and a hybrid `d2l.SeqLM` of the same width whose first block is Gated DeltaNet and second block attention, as in the recall experiment. With this recipe, Gated DeltaNet alone reached a perplexity of about 3.1.

```
data = d2l.PennTreebank(batch_size=64, num_steps=128)
```

```
models = {'attention (TinyLM)': d2l.TinyLM(len(data.vocab), d_model=128,
                                           num_heads=4, num_blks=2, max_len=128),
          'hybrid': d2l.SeqLM(len(data.vocab), 128, [
              d2l.GatedDeltaNet(128, 4), ConvAttention(128, 4)])}
for name, model in models.items():
    print(f'{name:20s} perplexity {d2l.fit_lm(model, data):.2f}')
```

```
attention (TinyLM)   perplexity 3.27
hybrid               perplexity 3.17
```

The hybrid reaches a perplexity of about 3.15, close to Gated DeltaNet alone, and the transformer between about 3.25 and 3.4, depending on the run. Language modeling at this scale barely registers the difference in recall, because predicting the next character of ordinary text rarely requires looking up a pair stored long before. A comparison by perplexity alone can therefore miss what a model cannot recall.

12.6.3 Inference Memory

During generation a model keeps, for every sequence it is serving, what it needs to compute the next token. For an attention layer this is the key–value cache of Section 11.3: one key

and one value per past token for each of the n_{kv} key–value heads of dimension d_{head}, that is,

$$2 \cdot n_{\text{kv}} \cdot d_{\text{head}} \cdot (\text{bytes per number})$$

bytes per token and layer. With grouped-query attention, 8 key–value heads of dimension 128 in 16-bit numbers, that is 4 KB per token. A recurrent layer keeps its state, whose size does not depend on the length: a Mamba-2 layer in a model of width 4,096 has 8,192 channels with 128 states each, about one million numbers, or 2 MB. The attention layer's cache overtakes this state after about 500 tokens, and at 128,000 tokens it holds 512 MB, 256 times as much.

Only the attention layers of a hybrid have caches that grow, so its memory per sequence is the transformer's times the fraction of layers that are attention, plus a constant. This is the accounting of windowed layers interleaved with global ones in Section 10.5, with a state of fixed size in place of each window's cache. Figure 12.10 shows the effect for a model of 32 layers. Jamba (Lieber et al. 2024), with 4 attention layers among 32, reports a cache of 4 GB at a context of 256,000 tokens, against 32 GB for a transformer of similar size. Memory per sequence decides how many sequences a GPU can serve at once, and because generating a token reads the whole cache from memory (Section 11.3), it also sets how fast each token is generated.

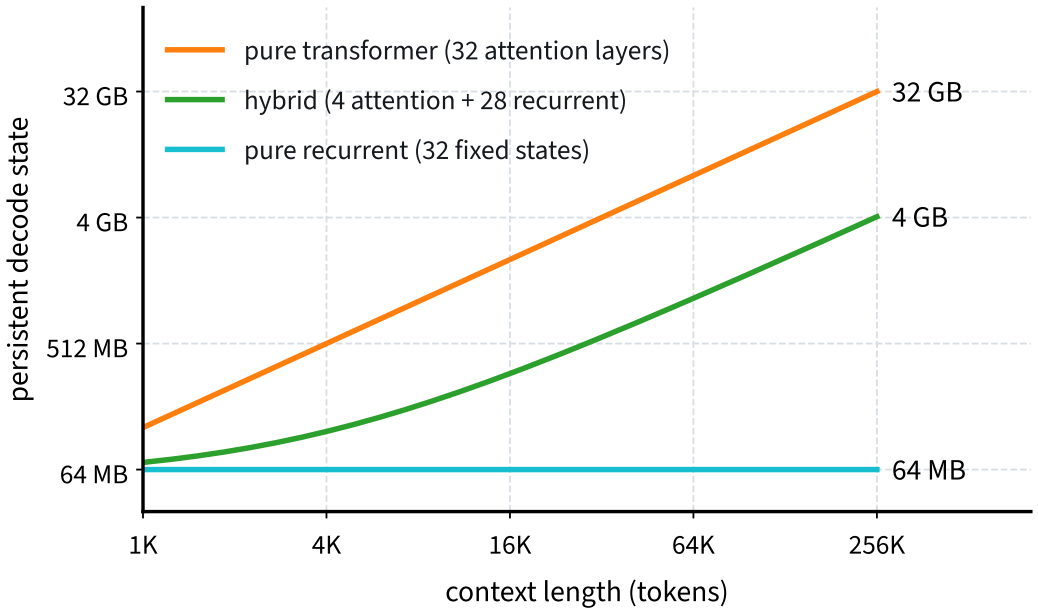


Figure 12.10: Memory held per sequence during generation, against context length, for a 32-layer model with grouped-query attention (8 key–value heads of dimension 128) and Mamba-2-sized recurrent states, in 16-bit numbers. A transformer's cache reaches 32 GB at 256K tokens; a purely recurrent model stays at 64 MB at every length; a hybrid with 4 attention layers pays an eighth of the transformer's cache.

The chapter asked how to build a memory of fixed size that trains in parallel, forgets the right things, stores many associations, and can overwrite them. Its answers, gated linear

recurrences computed by scans, selective state space models, matrix memories trained in chunks, and the delta rule, make the fixed state as useful as possible. The answer of practice adds a small memory that grows: a few attention layers, which retrieve exactly what the fixed state cannot hold. How few, where to place them, and whether some of them can attend to a window of recent tokens (Section 10.5) instead of the whole context are questions that current models answer differently; Section 12.7 lists some of their choices.

12.6.4 Summary

A state of fixed size holds a fixed number of bits, so a recurrent model can recall only as many items from its context as its state can store, while an attention layer keeps every token and can always look back. A matrix memory, which writes by adding outer products and reads linearly, holds at best about one value per number of its state: every key needs a slot of its own. When the task can ask about more keys than the state has slots, keys that share a slot overlap, by at least Welch's bound, and recall falls as the number of stored pairs grows. On multi-query associative recall, Gated DeltaNet recalls nearly every pair when there are few and about three quarters of 64, and gated linear attention, whose writes only add, recalls fewer; both curves still rise with longer training. Without the short convolution, which brings each key together with its value, both recall far less. A hybrid whose second layer is attention recalls every pair, and so does attention in almost every run. Language-modeling perplexity barely shows the difference. During generation only the attention layers keep a cache that grows with the context, so a hybrid needs the transformer's cache times its fraction of attention layers, an eighth for a model with one attention layer in eight. Most recent large models with recurrent layers are hybrids for this reason.

12.6.5 Exercises

1. **The counting bound.** A model reads a string of k tokens from a vocabulary of V and must then reproduce it.
   1. Prove Equation 12.33 for a model whose state between reading and writing has N numbers of b bits and whose output is a deterministic function of that state.
   2. Suppose instead that the model may output a wrong string with probability ϵ when the string is drawn uniformly. Show that $Nb \geq (1-\epsilon)\, k \log_2 V - 1$. (Hint: count the strings that each state can decode correctly.)
   3. How many tokens of a vocabulary of 50,000 can a model with 48 layers of Mamba-2 state, 2 MB each in 16-bit numbers, copy at most? Why does no model of this size come close in practice?
2. **Memory for many users.** A GPU with 80 GB serves a model of 32 layers, with the cache sizes of Figure 12.10, after 16 GB of the GPU's memory are taken by the weights.
   1. How many sequences of 128,000 tokens can the GPU serve at once with the transformer, the hybrid with 4 attention layers, and the recurrent model?
   2. Grouped-query attention with fewer key–value heads, and 8-bit numbers for the cache, each halve the cache. Apply both to the transformer and to the hybrid, and repeat the count. Which change matters more for the hybrid?
3. code **Keys and slots.** The recurrent models of the recall sweep have 256 slots and the task draws keys from 512.
   1. Train Gated DeltaNet for 16,000 and for 32,000 steps, once with keys drawn from 512 and once from 128, fewer than the slots. Where do the four curves differ? Which differences come from training time and which from the number of slots?

2. Keep 512 keys and double the width to 64, with eight heads. How many slots does the model have now, and how does its recall curve move?
3. Compute the bound Equation 12.34 for a row of dimension 4 that hosts 8 possible keys, and estimate how often two stored keys share a row in a sequence of 64 pairs.

4. code **Where and how much attention?**
   1. Train the hybrid `AG`, with attention in the first block, and compare its recall with that of `GA`. Does the position of the attention layer matter? Consider which layer can bring each key together with the value that follows it.
   2. Train four-block models `GGGG`, `GGGA`, and `GAGA` of the same width. Do two more recurrent layers add as many slots as the section's count predicts, and does one attention layer among four still recall every pair?
5. code **A window instead.** Replace the attention mixer by attention that sees only the last w tokens, by masking the scores of earlier tokens.
   1. Train the hybrid with $w = 32$ and $w = 128$ and compare recall at every number of pairs. Which queries can a window of 32 answer, and which must the recurrent layers answer?
   2. How large is the cache of this layer during generation? What does the comparison suggest about combining windows with recurrence?
6. code **Recall in text.** In the Penn Treebank validation text, mark the positions whose character completes a string of at least eight characters that already occurred earlier in the same window. Compare the loss of Gated DeltaNet, the hybrid, and `d2l.TinyLM` at the marked positions and at all other positions. Does the hybrid's advantage on recall show up where text repeats itself?

12.7 Further Reading

The sections of this chapter teach one representative of each idea. This section points to topics they had no room for: the theory behind the initialization of state space models, other ways to build and write to a fixed-size state, results on what such models can compute, other ways to combine recurrence and attention, and applications beyond language. Each topic states the problem it addresses and what it changes relative to the chapter. Implementations, tutorials, and courses for the material that the chapter does teach are listed in its introduction, Chapter 12. Nothing else in the book depends on this section.

12.7.1 State Space Models

12.7.1.1 Memory by Polynomial Projection

Section 12.2 initialized the rates of S4D as $\lambda_n = -(n+1)$ and gave one sentence of justification. The full argument asks what a fixed-size state should remember about an input $x(s)$, $s \le t$, that keeps growing. HiPPO (Gu et al. 2020) answers: the coefficients of the best approximation of the history by a polynomial of degree $N-1$, measured with a weight that spreads uniformly over the whole past. These coefficients obey a linear differential equation, $\mathbf{h}'(t) = \frac{1}{t}(\mathbf{A}\mathbf{h}(t) + \mathbf{B}x(t))$, whose matrix $\mathbf{A}$, the HiPPO matrix, is lower triangular and fixed by the choice of polynomials and weight. S4 (Gu, Goel, et al. 2022) dropped the factor $1/t$, initialized an SSM with this matrix, and made it trainable by writing it as a normal matrix plus a low-rank correction, which allows its convolution kernel to be computed quickly. S4D (Gu, Gupta, et al. 2022) uses a diagonal matrix instead: either the normal part without the correction, whose eigenvalues are complex, or the real rates $-(n+1)$ on the diagonal

of the HiPPO matrix itself, the choice of Section 12.2. Both work nearly as well as the full matrix. Gu et al. (2023) explain what the model without the factor $1/t$ remembers, a projection of the history onto polynomials under a weight that decays exponentially into the past, and interpret the convolution kernels of these models as projections onto orthogonal bases. Gu's thesis (Gu 2023) develops the whole line, including the control-theoretic view of SSMs as linear systems with inputs, states, and outputs.

12.7.1.2 Recurrences Without a Differential Equation

The continuous-time view is one way to arrive at stable, well-initialized linear recurrences, not the only one. The linear recurrent unit (Orvieto et al. 2023) starts from a plain diagonal recurrence with complex decays, parameterizes the decays so that their magnitude stays below one, initializes the magnitudes close to one, and normalizes the input so that the state does not grow. It matches S4 on long sequence benchmarks without discretization. S5 (J. T. H. Smith et al. 2023) replaces the d independent SSMs of an S4D layer by one SSM with vector input and output, computed with a parallel scan rather than a convolution, which is the route the selective models of Section 12.3 take.

12.7.1.3 Long Convolutions

The kernel of S4D in Section 12.2 is a sum of N decaying exponentials, one per state dimension, which is what gives the layer a recurrent form with a small state. H3 (Fu et al. 2023) kept such kernels and chained two SSMs, with elementwise products of input projections between them in the roles of queries, keys, and values. The first SSM only shifts its input, which pairs each token with its predecessor, and the second, diagonal SSM accumulates these pairs. A layer can then compare a token with earlier ones and recall what followed them, and H3 closed much of the gap to attention on language. Hyena (Poli et al. 2023) drops the constraint on the kernel: a small network maps each position to the kernel value there, the convolution over the whole sequence is computed with the FFT in $O(T \log T)$ time, and elementwise gates sit between the convolutions. Such kernels have no recurrent form with a small state, so generation either recomputes the convolution or approximates the kernel by a recurrence afterwards.

12.7.1.4 Kernels for Selective Scans

The selective scan of Section 12.3 produces a state of $H \times N$ numbers at every position, too much to write to GPU memory for long sequences. The Mamba kernel (Gu and Dao 2023) loads the inputs into fast on-chip memory, discretizes and scans there, writes only the outputs, and recomputes the states during the backward pass instead of storing them, the trade that Section 13.4 discusses as activation checkpointing. Mamba-2 (Dao and Gu 2024) restricts the decay to one scalar per head so that most of the work becomes the matrix products of Section 12.4, which GPUs perform much faster than elementwise scans. Mamba-3 (Lahoti et al. 2026) makes three changes to the selective model: a trapezoidal discretization, which combines the current and the previous input in each update instead of the current one alone; a complex-valued state, which lets the state rotate and track quantities such as parity; and a multi-input, multi-output update, which raises the arithmetic work per byte of state read during generation.

12.7.2 Writing to a Matrix Memory

12.7.2.1 Exponential Gates

The LSTM's gates are sigmoids, so a forget gate can keep at most all of the state and an input gate can write at most the full candidate. The mLSTM of xLSTM (Beck et al. 2024) uses a matrix memory with an exponential input gate, which lets a single important token outweigh everything stored before it, and keeps a second state, the sum of the gated keys, by which it divides each read. Exponentials overflow quickly, so the layer carries the logarithm of the largest gate seen so far and rescales both states by it, the same device that the online softmax of fused attention kernels uses.

12.7.2.2 Generalized Delta Rules

The delta rule of Section 12.5 removes from the memory the component along the key it is about to write, by the same amount β_t for every coordinate. RWKV-7 (Peng et al. 2025) decouples these choices: it decays each coordinate of the memory at its own rate, removes along one vector, and writes along another, so that its transition is no longer symmetric. DeltaProduct (Siems et al. 2025) applies several delta-rule steps per token, so that each transition is a product of several reflections or projections and can rotate the memory, which helps on tasks that require tracking a state through permutations.

12.7.2.3 Memories That Are Networks

In Section 12.5 the memory was a matrix fitted to the pairs of the current sequence by one gradient step per token. TTT layers (Y. Sun et al. 2024) replace the matrix by a small multilayer perceptron and fit it by gradient steps as the sequence is read. Titans (Behrouz et al. 2025) add momentum to these steps, so that a surprising token keeps influencing the next few updates, and a forgetting term like the decay of Gated DeltaNet. A network can represent nonlinear associations between keys and values that a matrix cannot, but whether this recalls more per byte of state than a matrix of the same size is a current research question.

12.7.2.4 Expressivity of Fixed-State Models

Some tasks are hard for the linear recurrences of this chapter, independently of memory size. Merrill et al. (2024) show that linear recurrences with diagonal transitions, like transformers, fall in a class of parallel computations that cannot track the state of a general finite automaton over arbitrary lengths, for instance the result of a long sequence of permutations. Nonlinear recurrent networks and recurrences with non-diagonal transitions can. Grazzi et al. (2025) show that allowing negative eigenvalues, as the delta rule with $\beta_t > 1$ does, is enough for parity and helps on related tasks, and DeltaProduct extends the range further.

12.7.3 Recurrence and Attention Together

12.7.3.1 Hybrid Models in Practice

Section 12.6 showed that one attention layer next to a recurrent one restores recall. Released models make different choices for the ratio and the kind of attention. Griffin (De et al. 2024) alternates a gated diagonal recurrence with the sliding-window attention of Section 10.5. Jamba (Lieber et al. 2024; Jamba Team 2024) uses one attention layer for every seven Mamba layers, together with mixture-of-experts MLPs. Waleffe et al. (2024) trained

matched transformers, Mamba models, and hybrids on the same data and found that hybrids with a small fraction of attention layers matched or exceeded the transformer on most tasks, including recall. Samba (Ren et al. 2024) combines Mamba layers with sliding-window attention, and Zamba (Glorioso et al. 2024) shares the weights of one attention block among several positions in the stack. More recent models build their recurrent layers on the delta rule: Qwen3-Next (Qwen Team 2025) interleaves Gated DeltaNet layers with full attention layers, three to one, and Kimi Linear (Kimi Team 2025b) does the same with a Gated DeltaNet variant whose decay acts on each key coordinate.

12.7.3.2 Between a Fixed State and a Full Cache

A recurrent layer stores a fixed amount, and attention stores every token. Two designs sit in between. Based (Arora, Eyuboglu, Zhang, et al. 2024) combines linear attention, whose feature map approximates the exponential of the softmax by its Taylor expansion, with exact attention over a short window of recent tokens. The size of the feature map and of the window then set how much the layer stores, and the authors trace how recall improves as the state grows. Log-linear attention (Guo et al. 2025) keeps one matrix state per block of the past, with blocks that double in length the further back they lie, so the memory grows with the logarithm of the length rather than staying fixed or growing linearly. The construction applies to the gated recurrences of this chapter, Mamba-2 and Gated DeltaNet among them.

12.7.3.3 Distillation from Transformers

Training a large recurrent model from scratch is as expensive as training a transformer. An alternative is to convert one: initialize the recurrent layers from the attention weights of a trained transformer, using the correspondence between linear attention and matrix-state recurrences of Section 12.4, and train the result to match the transformer's outputs. Junxiong Wang et al. (2024) do this to turn a transformer into a hybrid that keeps some of its attention layers, and Bick et al. (2024) distill in stages, first matching the mixing matrices of attention, then the hidden states, then the outputs.

12.7.4 Beyond Language

12.7.4.1 Images

An image has no natural order. If it is flattened into a sequence of patches and scanned causally, as a language model scans text, each patch sees only the patches above it and to its left. Vision Mamba (Zhu et al. 2024) scans the sequence of patches forward and backward and adds the two outputs. VMamba (Y. Liu et al. 2024) scans the grid of patches along rows and along columns, each in both directions, and merges the four results, so that every patch receives information from the whole image.

12.7.4.2 Audio

Raw audio has tens of thousands of samples per second, far too many for attention over a recording of any length. SaShiMi (Goel et al. 2022) stacks S4 layers in a multiscale network that pools the sequence to coarser rates and upsamples it again, and generates waveforms one sample at a time with the recurrent form of each layer.

12.7.4.3 Genomics

A stretch of DNA can regulate a gene tens of thousands of bases away, and a strand and its reverse complement encode the same information. HyenaDNA (Nguyen et al. 2023) trains Hyena on contexts of up to a million bases with one token per nucleotide. Caduceus (Schiff et al. 2024) builds on Mamba, scans in both directions, and makes each layer equivariant to taking the reverse complement, so that a sequence and its reverse complement receive consistent predictions.

Chapter 13

Computational Performance

Until now we have trained every model by writing the network, starting the training loop, and waiting. We took the time of a step and its memory as given. That stops working when a run takes days, a model does not fit on the card, or a second GPU makes training slower. Consider the GPT of Section 11.2. Dividing the arithmetic of one training step by the rate in the GPU's specification gives the time the step should take. On the GPU that runs this book's notebooks, the measured step takes several times longer. Where does the rest of the time go?

Computer architects met this question long before deep learning did. By the 1990s processors had become so much faster than their memory that many programs spent much of their time waiting for data. In 2009 Williams, Waterman, and Patterson drew the consequence as one picture, the *roofline* (Williams et al. 2009): a program runs no faster than its arithmetic units allow, and no faster than memory can deliver the bytes it needs. A GPU driven from Python adds a third limit, the cost of issuing each operation. Which of the three holds a step back decides which change speeds it up. This view explains choices that earlier chapters made, such as large minibatches and 16-bit numbers (Section 5.5). It also gives a method: measure a program, find the resource that limits it, relieve that resource, and measure again.

Section 13.1 asks which of the three resources limits an operation, and how to time GPU code correctly. Section 13.2 asks where a machine's arithmetic rate and bandwidth come from, and why their ratio tends to grow. Section 13.3 asks how a compiler removes memory traffic and launch overhead that eager execution cannot avoid. Section 13.4 asks where the memory of a training step goes, and how a model of a billion parameters can train on one card. Section 13.5 builds data parallelism by hand and asks when a second GPU shortens a step. Section 13.6 asks how production libraries hide communication behind computation and divide a model that no single GPU holds. Section 13.7 asks how far the whole method takes the training step of a GPT.

The chapter stays within one machine with up to four GPUs, and it uses the kernels that the frameworks provide instead of writing new ones (Section 5.4). Training across machines, serving engines, and quantization for inference belong to the Language Models part.

Resources and Further Reading

The resources below accompany the sections of this chapter; each names the section it goes with. All are freely accessible online. Related topics that the chapter does not teach, such as other accelerators, kernels written by hand, compressed optimizer state, and collective algorithms beyond the ring, are collected in Section 13.8.

Books and long-form texts

- How to Scale Your Model — Austin et al., Google DeepMind (2025) — rooflines, sharding notation, and the cost of collectives worked out for TPUs and GPUs in JAX; the arithmetic of Section 13.1 and the cost model of Section 13.5, carried on to clusters of thousands of chips.
- The Ultra-Scale Playbook — Hugging Face (2025) — the memory budget of a training step, data parallelism, and ZeRO sharding, each with measurements, from Section 13.4 through Section 13.6, and on to parallelism across many machines.
- Making Deep Learning Go Brrrr From First Principles — Horace He (2022) — the division of a program's time into compute, memory bandwidth, and overhead that Section 13.1 measures, in a short essay by one of the authors of `torch.compile`.

Courses and lectures

- Stanford CS336: Language Modeling from Scratch — assignments that profile a transformer, apply compilation and mixed precision, and train it on several GPUs, the work of Section 13.1 through Section 13.7 as graded problems; lectures on YouTube.
- CMU 15-442/15-642: Machine Learning Systems — a full course on the hardware, compilers, and parallelism of Section 13.2 through Section 13.6, with lecture notes and assignments.
- GPU MODE lecture series — CUDA, Triton, and the internals of fused attention kernels, for readers who want to write the kernels that Section 13.3 only generates or calls.

Foundational papers

- Roofline: An Insightful Visual Performance Model for Multicore Architectures — Williams, Waterman & Patterson (2009) — the original roofline and its ridge point, the model of Section 13.1.
- PyTorch 2: Faster Machine Learning Through Dynamic Python Bytecode Transformation and Graph Compilation — Ansel et al. (2024) — how TorchDynamo captures a graph from Python bytecode and Inductor compiles it, the PyTorch side of Section 13.3.
- Mixed Precision Training — Micikevicius et al. (2018) — fp32 master weights and loss scaling, the recipe of Section 13.4.2, and the reason fp16 needs a scaler where bf16 does not.
- Training Deep Nets with Sublinear Memory Cost — Chen et al. (2016) — recomputing activations in the backward pass instead of storing them, the trade of Section 13.4.3, with a placement of the stored layers that makes memory grow as the square root of the depth.
- PyTorch Distributed: Experiences on Accelerating Data Parallel Training — Li et al. (2020) — gradient buckets and the overlap of the allreduce with the backward pass, the design of DDP in Section 13.6.
- ZeRO: Memory Optimizations Toward Training Trillion Parameter Models — Rajbhandari et al. (2020) — the three stages of sharding optimizer state, gradients, and weights of Section 13.6.3.

Tutorials and notes

- PyTorch Performance Tuning Guide and Automatic Mixed Precision recipe — a checklist of settings that Section 13.2 and Section 13.4 explain, from pinned memory to autocast.
- torch.compile tutorial, JAX: just-in-time compilation, and JAX: sharded computation — the framework documentation behind Section 13.3 and Section 13.6, including graph breaks, recompilation, and sharded arrays.
- Understanding GPU Memory — PyTorch blog (2023) — recording and reading memory snapshots with `torch.cuda.memory._record_memory_history`, the tool of Section 13.4.1.4.
- How to Train Really Large Models on Many GPUs? — Lilian Weng (2021) — a survey of data, pipeline, and tensor parallelism and of ZeRO, the background of Section 13.5 and Section 13.6.

13.1 The Performance Model

Every network we have trained so far spends most of its arithmetic in matrix products, and we have assumed that a GPU computes them at the rate its specification states. The assumption is often wrong. On the RTX 4090 that runs this book's notebooks, a product of two 8192×8192 matrices in bf16 comes close to the 165 TFLOP/s that the specification promises. A product of two 256×256 matrices reaches less than two percent of it, although the operation and the chip are the same and both products are timed correctly. A training step is a long sequence of operations, many of them small. If most of them run at a few percent of the peak, a faster GPU or a second one changes little, because the time goes to waiting rather than to computing. Why does the same operation run at such different speeds, and how do we tell which case a program is in?

This section explains the gap. Every operation on a GPU is limited by one of three resources: the arithmetic units, the memory bandwidth that supplies them with data, or the fixed cost of issuing the operation from Python. A single number computed from the shapes of an operation, its arithmetic intensity, tells which of the first two applies. Only measurement reveals the third. Together they give a method in four steps, which the rest of this chapter follows. We **measure** what a program does, **classify** which resource limits it, **fix** that resource with the matching technique, and **re-measure**, because each fix moves the bottleneck somewhere else.

```
import re
import time
```

```
# Run fp32 matrix products on the tf32 tensor cores (PyTorch's default is
# full fp32). The bf16 products below are unaffected.
torch.set_float32_matmul_precision('high')
```

13.1.1 Arithmetic Intensity and the Roofline

Before we time anything, we can predict from the shapes of an operation which resource should limit it. The prediction needs two counts: the arithmetic the operation performs and the bytes it moves.

13.1.1.1 Counting FLOPs and Bytes

Every tensor operation makes two demands on a GPU. It performs arithmetic, which we count in *floating-point operations* (FLOPs), one addition or one multiplication each. And it moves bytes: its inputs come from the GPU's memory, and its outputs go back there. The GPU has a separate budget for each demand. The RTX 4090 performs at most $P = 165 \times 10^{12}$ FLOP/s in bf16 on its tensor cores, and its memory delivers at most $\beta = 1.008 \times 10^{12}$ bytes/s. Which budget runs out first depends on the ratio of the two demands, the *arithmetic intensity*

$$I \;=\; \frac{\text{FLOPs performed}}{\text{bytes moved}}.$$

An operation of high intensity performs many FLOPs on each byte it fetches. An operation of low intensity performs only a few, or less than one.

The most important operation in deep learning is the matrix product $\mathbf{XW}$ with $\mathbf{X} \in \mathbb{R}^{B\times D}$ and $\mathbf{W} \in \mathbb{R}^{D\times F}$, a linear layer applied to a minibatch of B examples. It performs $2BDF$ FLOPs, one multiplication and one addition for each of the BDF terms of the sums. At best it reads $\mathbf{X}$ and $\mathbf{W}$ once and writes the result once, which moves $BD + DF + BF$ numbers. With b bytes per number, its intensity is

$$\frac{2BDF}{b\,(BD + DF + BF)} \;=\; \frac{2}{b}\left(\frac{1}{F} + \frac{1}{B} + \frac{1}{D}\right)^{-1} \;\approx\; \frac{2B}{b} \quad \text{for } B \ll D, F. \tag{13.1}$$

The middle form shows that the smallest of the three dimensions limits the intensity. The approximation covers the common case of a minibatch that is narrower than the layer. There the intensity grows linearly with the batch size, because every weight fetched from memory is used once for each row of the batch. A batch of one uses each weight once and discards it.

A GPT with about a billion parameters has attention projections of width $D = F = 2048$. In training, a minibatch of 8 sequences of 1,024 tokens gives $B = 8192$, and the exact form of Equation 13.1 gives about 910 FLOP/byte in bf16 ($b = 2$). Here the width, not the batch, is the smallest dimension. When the same model generates text one token at a time, $B = 1$ and the intensity is about 1 FLOP/byte. The same layer, with the same weights, spans a factor of about a thousand in intensity between the two uses.

An elementwise operation lies at the low extreme. The operation $y_i = x_i + 1$ performs one FLOP per element and moves $2b$ bytes, since it reads x_i and writes y_i. Its intensity is $1/(2b)$, or 0.25 FLOP/byte in bf16, however large the tensor.

13.1.1.2 The Roofline Model

The *roofline model* (Williams et al. 2009) turns intensity into a bound on speed. An operation cannot run faster than the arithmetic units allow, and it cannot run faster than memory delivers its bytes, which caps it at $I\beta$ FLOP/s. The attainable throughput is therefore

$$\text{performance} \;\le\; \min(P,\; I\beta). \tag{13.2}$$

On log–log axes the bound is a sloped line that meets a flat roof (Figure 13.1). The two meet at the *ridge point* $I^* = P/\beta$. An operation of intensity below I^* is *bandwidth-bound*: memory cannot deliver bytes fast enough to keep the arithmetic units busy, so additional arithmetic on each byte costs no time. An operation above I^* is *compute-bound*: the arithmetic units are saturated, and moving fewer bytes would not make it faster. The elementwise operation $x + 1$ in bf16 has an intensity of 0.25 FLOP/byte, so it runs at most at 0.25β, about a quarter of a trillion FLOP/s. That is less than a six-hundredth of the compute roof.

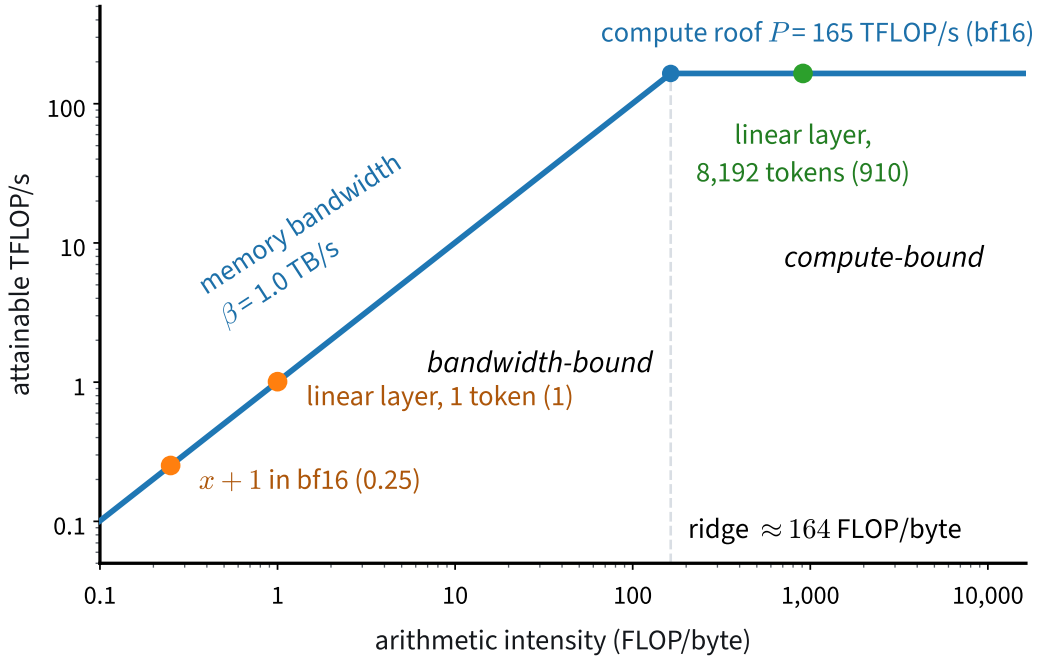


Figure 13.1: The roofline of the RTX 4090. Attainable throughput is the smaller of the compute roof P and the arithmetic intensity times the memory bandwidth β. The points mark the examples of the text: $x + 1$ in bf16, a linear layer of width 2048 applied to one token, and the same layer applied to a minibatch of 8,192 tokens. The ridge point lies near 164 FLOP/byte.

13.1.1.3 The Ridge Point of the RTX 4090

The peak throughput of the tensor cores cannot be queried from the device, so we take P and β from the specification sheet and print the name of the GPU that the code runs on. On another GPU, replace the two constants with the values from its specification.

```
print(torch.cuda.get_device_name())
```

```
NVIDIA GeForce RTX 4090
```

```
peak_tflops = 165.2    # RTX 4090: dense bf16 on the tensor cores
bandwidth_tbs = 1.008  # RTX 4090: GDDR6X memory
```

```
print(f'ridge point = {peak_tflops / bandwidth_tbs:.0f} FLOP/byte')
```

```
ridge point = 164 FLOP/byte
```

A ridge point of 164 FLOP/byte means that an operation must perform about 164 FLOPs for every byte it moves before the arithmetic units, rather than memory, set its speed. On other current GPUs the ridge lies at about 120 FLOP/byte (RTX 5090) or about 300 (H100, B200). Only large matrix products reach these values. Elementwise operations, normalizations, and the generation of text one token at a time all lie far below the ridge, and most of this chapter is therefore about moving fewer bytes.

For a square product with $B = D = F = n$ in bf16 the intensity is $n/3$, so the roofline model declares every such product with n above about 500 compute-bound. To see whether the GPU agrees, we first need a way to measure it.

13.1.2 Timing GPU Code

Python does not wait for the GPU. As Section 5.7.2.1 showed, each operation is placed in the device's queue and Python moves on: 32 products of 4096×4096 matrices are issued in a small fraction of the time the GPU needs to compute them. A clock read right after a call therefore measures how long Python took to queue the work, not how long the GPU took to do it (Figure 13.2). A correct measurement runs the operation once before timing it, waits for the device before it starts and before it stops the clock, and divides the time of many repetitions by their number.

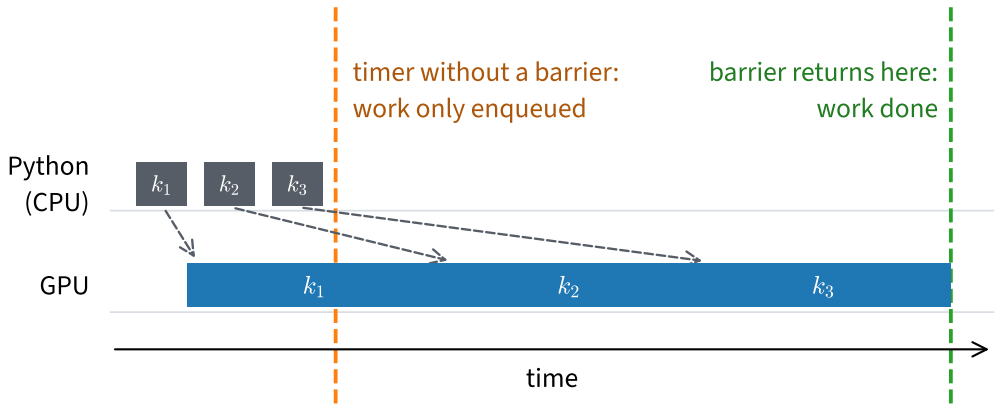


Figure 13.2: Asynchronous dispatch. Python enqueues kernels and moves on, and the GPU executes them behind it. A timer read without a barrier stops when the last kernel has been *enqueued*; only a synchronization point waits until the work is *done*.

13.1.2.1 Synchronization Points

Python waits for the device only when it needs a value that the device computes: to convert it into a Python number or a NumPy array, to print it, or to branch on it, as in `if loss < 0.1:`. Each such *synchronization point* drains the queue up to that value. One kind is easy to miss: an operation whose output *shape* depends on the data, such as selecting the positive entries `x[x > 0]`, waits because Python must learn the shape of the result before it can continue.

After a synchronization point the device sits idle until Python sends more work. A training loop that reads its loss on every step therefore drains the queue once per step. To see what this costs, we time two loops of 1,000 steps. Each step multiplies a 1024×1024 matrix by itself and adds the sum of the result to a running total on the device. This work takes the GPU some tens of microseconds, comparable to the time Python needs to issue it. The two loops differ only in when they read the total: the first converts it into a Python number after every step, the second once at the end.

```
x = torch.ones(1024, 1024, device=d2l.try_gpu())
s = torch.zeros((), device=x.device)
s += (x @ x).sum()
float(s)                          # Warm up: load the kernels

torch.cuda.synchronize()
t0 = time.perf_counter()
for _ in range(1000):
    s += (x @ x).sum()
    total = float(s)              # Reads the value: a barrier every step
read_every = time.perf_counter() - t0

torch.cuda.synchronize()
t0 = time.perf_counter()
for _ in range(1000):
    s += (x @ x).sum()            # Stays on the device: no barrier
total = float(s)                  # One read at the end
read_once = time.perf_counter() - t0
print(f'read every step: {read_every:.3f} s   read once: {read_once:.3f} s')
```

```
read every step: 0.064 s   read once: 0.040 s
```

`torch.cuda.synchronize()` waits until the device has finished all queued work, so that each loop starts on an idle device. `float(s)` converts a tensor with one element into a Python number, which waits for its value.

Reading the value on every step makes the loop slower, although the device does the same work. Without reads, host and device work at the same time: while the GPU computes one step, Python issues the next, and each step costs the longer of the two times. A read on every step makes them take turns. The device idles while the host waits for the value and then issues the next step, and the host idles while the device computes. Each step then costs the host time plus the device time plus the latency of the round trip.

Here the reads make the loop about one and a half times as slow.

A training loop should therefore read values from the device once per logging interval, not once per step, and keep running sums on the device. The `d2l.Trainer` follows this rule: its training and validation steps return losses as device arrays, which it adds up on the device and converts into Python numbers once per epoch.

13.1.2.2 A Benchmark Helper

Every timing in this chapter follows the same rules, which `d2l.Benchmark` of Section 5.7.2.1 packages. It calls a function a few times to warm up, which excludes one-time costs such

as loading kernels, selecting algorithms, and compiling. It then waits for the device, times `repeats` further calls, waits again, and stores the average time per call.

On a machine without a GPU, PyTorch runs operations synchronously, and `Benchmark` needs no barrier.

We time the product of two 4096×4096 matrices in bf16.

```
a = torch.randn(4096, 4096, device=d2l.try_gpu(), dtype=torch.bfloat16)
d2l.Benchmark(lambda: torch.mm(a, a), desc='4096 x 4096 product')
```

```
4096 x 4096 product: 0.89 ms/call
```

Each product takes a little under a millisecond. It performs $2 \cdot 4096^3 \approx 1.4 \times 10^{11}$ FLOPs, so the GPU computes at more than 150 TFLOP/s, over 90% of its specification.

13.1.3 Three Regimes

The time of a GPU program divides among compute-bound kernels, bandwidth-bound kernels, and idle gaps in which the device waits for Python to send work (He 2022). Whichever part dominates names the program's *regime*. Two experiments locate the three regimes on the RTX 4090: matrix products of growing size, and elementwise operations on a large tensor.

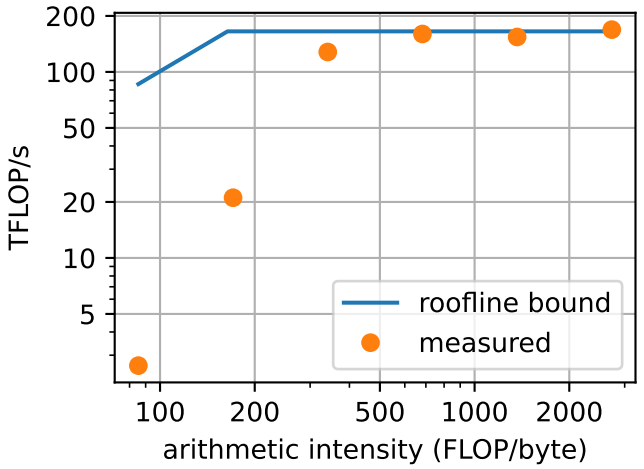
13.1.3.1 Matrix Products across Sizes

Does a square product reach the roof as soon as the roofline says it should, near $n = 500$? We time products in bf16 from $n = 256$ to $n = 8192$. Each size runs a hundred times, so that the latency of the final wait is spread over many calls.

```
sizes = [256, 512, 1024, 2048, 4096, 8192]
times = []
for n in sizes:
    a = torch.randn(n, n, device=d2l.try_gpu(), dtype=torch.bfloat16)
    times.append(d2l.Benchmark(lambda: torch.mm(a, a), repeats=100).time)
```

We convert each time into TFLOP/s and plot it against the intensity $n/3$, together with the roofline bound Equation 13.2, which has its corner at the ridge point.

```
measured = [2 * n**3 / t / 1e12 for n, t in zip(sizes, times)]
intensity = [n / 3 for n in sizes]  # FLOP/byte of a square bf16 product
ridge = peak_tflops / bandwidth_tbs
roof_x = [intensity[0], ridge, intensity[-1]]
roof_y = [min(peak_tflops, bandwidth_tbs * i) for i in roof_x]
d2l.plot([roof_x, intensity], [roof_y, measured],
         'arithmetic intensity (FLOP/byte)', 'TFLOP/s',
         legend=['roofline bound', 'measured'],
         xscale='log', yscale='log', fmts=('-', 'o'))
```



Printing the time per call of each size shows where the measurements fall short of the bound.

```
for n, t, tf in zip(sizes, times, measured):
    print(f'n = {n:4d}: {1e6 * t:7.1f} us per call, {tf:5.1f} TFLOP/s')
```

```
n =  256:    12.7 us per call,   2.6 TFLOP/s
n =  512:    12.7 us per call,  21.1 TFLOP/s
n = 1024:    16.8 us per call, 128.0 TFLOP/s
n = 2048:   107.4 us per call, 160.0 TFLOP/s
n = 4096:   891.7 us per call, 154.1 TFLOP/s
n = 8192:  6513.3 us per call, 168.8 TFLOP/s
```

The roofline bound is flat from $n = 512$ on, but the measurements reach it only at the largest sizes. From $n = 2048$ on, the GPU computes within about 15% of its specification: these products are compute-bound in fact as well as in the model. A measurement slightly above the specification is possible, because the specification assumes the card's rated clock of 2.52 GHz and the card often runs faster.

The small sizes behave differently. From $n = 256$ to $n = 1024$ the work grows 64-fold, but the time per call changes by less than a factor of two, and each doubling of n multiplies the throughput by five to ten. A time per call that barely depends on the size cannot be spent on arithmetic. It is the time h that the host needs to issue one call: Python, the framework, and the driver that launches the kernel. When calls are queued back to back, host and device work at the same time, and each call takes the longer of the two:

$$t_{\text{call}} \approx \max\left(h, \frac{2n^3}{P}\right).$$

For small n the device finishes each product before the host has issued the next one, and the throughput $2n^3/h$ rises with the cube of n. This is the third regime, *overhead-bound*. The knee of the curve lies where the device time of a product overtakes the host time, $n \approx (hP/2)^{1/3}$: for $h = 20\,\mu$s and the peak of the RTX 4090, near $n = 1200$. The time per call at the smallest sizes in the output above is the value of h for this framework and this machine.

The roofline has no term for overhead. It is a ceiling, and the measured points meet it only for operations large enough that the host time per call is negligible. The faster the GPU, the larger an operation must be to get there: $n \approx (hP/2)^{1/3}$ grows with P.

A product can also fall short of the roof for a reason inside the device. A GPU executes a kernel on its *streaming multiprocessors* (SMs), the independent processors of the chip; the RTX 4090 has 128 of them (Section 13.2.3). A matrix-product kernel splits the output into tiles and gives each SM one tile at a time. A product whose output has fewer tiles than the GPU has SMs leaves some SMs idle, however fast the host issues it. The share of the peak that it can reach is then at most the share of SMs that hold a tile.

13.1.3.2 Elementwise Operations

Products large enough to escape the overhead regime lie above the ridge point, so the sweep never shows the bandwidth-bound regime. To see it, we apply single elementwise operations of very different arithmetic cost to a tensor of 8192×8192 numbers in fp32, 256 MiB: an addition, a multiplication, a sine, and a sigmoid. Each reads the tensor once and writes a result of the same size.

```
x = torch.randn(8192, 8192, device=d2l.try_gpu())  # 256 MiB in fp32
for desc, f in [('add', lambda x: x + 1), ('mul', lambda x: x * 1.5),
                ('sin', torch.sin), ('sigmoid', torch.sigmoid)]:
    print(d2l.Benchmark(lambda: f(x), desc=desc))
```

```
add: 0.59 ms/call
mul: 0.59 ms/call
sin: 0.59 ms/call
sigmoid: 0.58 ms/call
```

All four operations take the same time, a little over half a millisecond. Each moves 512 MiB, 5.4×10^8 bytes, which amounts to about 0.9 TB/s, close to the 1.008 TB/s of the specification. A sine costs many more FLOPs than an addition, yet it runs no slower, because the bytes each kernel moves set its time. This is the defining behavior of a bandwidth-bound operation: additional arithmetic costs no time until the intensity approaches the ridge point.

Activation functions and normalization layers chain several elementwise operations. The function below is the tanh approximation of the GELU activation $z\,\Phi(z)$ (Section 4.1.4) with rounded constants, and it applies eight elementwise operations to its input. Executed one operation at a time, each operation is a kernel of its own, which reads its inputs from memory and writes its result back.

```
def gelu_ish(x):
    """The tanh approximation of GELU with rounded constants: a chain of
    eight elementwise operations."""
    return 0.5 * x * (1 + torch.tanh(0.8 * (x + 0.04 * x**3)))

print(d2l.Benchmark(lambda: x + 1, desc='1 op'))
print(d2l.Benchmark(lambda: gelu_ish(x), desc='chain'))  # Eager: 8 kernels
```

```
1 op: 0.59 ms/call
chain: 5.24 ms/call
```

The chain takes eight to nine times as long as the single addition, and the bytes explain the factor. The addition reads one tensor and writes one. The eight operations of the chain read and write 18 tensors of the same size between them. Each of the six operations with one tensor input moves two; the sum $x + 0.04x^3$ and the final product combine two tensors and move three each. One kernel that read `x` once and wrote the result once would move 2 tensors instead of 18. Merging operations into such a kernel is called *fusion*, and a compiler can do it automatically.

13.1.3.3 Diagnosis and Fixes

The experiments have located the three regimes on one GPU (Figure 13.3). Each regime has a symptom by which we recognize it and a standard fix, summarized in Table 13.1. The later sections of this chapter develop each fix.

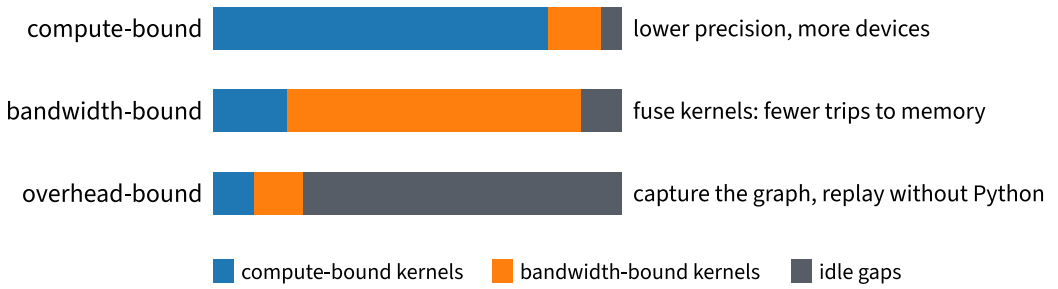


Figure 13.3: The three regimes. Where the time of a step goes, among compute-bound kernels, bandwidth-bound kernels, and idle gaps, determines which fix applies.

Table 13.1: Symptoms and standard fixes of the three regimes.

regime	symptom	standard fix
compute-bound	device busy; intensity above the ridge; time grows with the FLOPs	a format with faster tensor cores; more devices
bandwidth-bound	device busy; intensity below the ridge; time grows with the bytes moved	fusion; recomputation instead of storage; algorithms that reuse data on chip
overhead-bound	device idle between kernels; time per call independent of size	fewer and larger calls; capturing the whole program and replaying it without Python

The FlashAttention kernel of Section 10.5 is an example of the third fix for bandwidth: it computes attention in tiles held on the chip, so the $T \times T$ matrix of attention scores never reaches memory. A wrong diagnosis wastes effort. More FLOP/s do not speed up a bandwidth-bound operation, and a faster GPU does not help a program whose device sits idle.

13.1.4 Profiling a Training Step

A training step runs hundreds of kernels from all three regimes. A *profiler* records every kernel the device runs, with its duration, so that we can add up where the time of a step goes: how much each kernel takes, and how long the device sits idle.

13.1.4.1 Recording the Kernels

We profile one SGD step of an MLP with two hidden layers of width 4096 on a minibatch of 1,024 examples, in fp32, with the matrix products on the tf32 tensor cores. First we measure the wall-clock time per step with `Benchmark`.

```
net = nn.Sequential(nn.Linear(4096, 4096), nn.ReLU(),
                    nn.Linear(4096, 4096), nn.ReLU(),
                    nn.Linear(4096, 10)).to(d2l.try_gpu())
X = torch.randn(1024, 4096, device=d2l.try_gpu())
opt = torch.optim.SGD(net.parameters(), lr=0.01)

def step():
    opt.zero_grad()
    net(X).mean().backward()
    opt.step()

wall = d2l.Benchmark(step).time
```

`kernel_times` runs a function ten times under the profiler and returns the device time per call of every kernel that the function launched. We save it, and the printout below, for the case study at the end of the chapter.

```
def kernel_times(f, steps=10):
    """Device milliseconds per call of f, for every kernel that f runs."""
    with torch.profiler.profile(
            activities=[torch.profiler.ProfilerActivity.CUDA],
            acc_events=True) as prof:
        for _ in range(steps):
            f()
        torch.cuda.synchronize()
    return {e.key: e.device_time_total / 1e3 / steps
            for e in prof.key_averages()
            if e.device_type == torch.autograd.DeviceType.CUDA}
```

`prof.key_averages()` groups the recorded events by name. Kernels are the events that ran on the device; the other events, such as the calls that launch kernels, ran on the CPU. `prof.export_chrome_trace('trace.json')` writes the full timeline, which https://ui.perfetto.dev displays.

13.1.4.2 Reading the Profile

`show_profile` prints the device time per step against the wall-clock time, the share of the matrix products, and the kernels that take the most device time. It strips the return type, the namespaces, and the argument list from the C++ names of the kernels.

```
def show_profile(kernels, step_ms, rows=6):
    """Busy device time against the wall-clock time step_ms of a step, the
    share of the matrix products, and the kernels with the most time."""
    busy = sum(kernels.values())
    products = sum(ms for name, ms in kernels.items()
                   if re.search('gemm|xmma|cutlass|matmul|sdpa', name, re.I))
    print(f'wall clock {step_ms:.2f} ms per step, device busy {busy:.2f} ms '
          f'({busy / step_ms:.0%}), matrix products {products / busy:.0%}')
    for name, ms in sorted(kernels.items(), key=lambda kv: -kv[1])[:rows]:
        name = re.sub(r'^void |\(anonymous namespace\)::|\w+::|\(.*', '', name)
        print(f'{ms:6.2f} ms {ms / busy:4.0%}  {name[:52]}')
```

We profile the training step:

```
show_profile(kernel_times(step), 1000 * wall)
```

```
wall clock 2.61 ms per step, device busy 2.59 ms (99%), matrix products 80%
  0.83 ms  32%  Kernel2<cutlass_80_tensorop_s1688gemm_256x128_16x3_n
  0.80 ms  31%  Kernel2<cutlass_80_tensorop_s1688gemm_128x128_32x3_t
  0.42 ms  16%  multi_tensor_apply_kernel<TensorListMetadata<2>, Bin
  0.39 ms  15%  Kernel2<cutlass_80_tensorop_s1688gemm_128x256_16x3_n
  0.06 ms   2%  vectorized_elementwise_kernel<4, BinaryFunctor<float
  0.02 ms   1%  vectorized_elementwise_kernel<4, launch_clamp_scalar
```

The device is busy for almost all of the wall-clock time, so the step is not overhead-bound. Three quarters or more of the device time goes to a handful of matrix-product kernels; their names contain *gemm*, the traditional name for a general matrix product. These kernels are compute-bound: a product of shapes 1024×4096 and 4096×4096 has an intensity of about 340 FLOP/byte in fp32, against a ridge point of about 82 for tf32 (82.6 TFLOP/s over 1.008 TB/s).

The remaining kernels are elementwise kernels and reductions: the ReLUs, the bias gradients, which sum over the batch, and the SGD update. The update reads every weight and its gradient and writes the weight back, 12 bytes for two FLOPs. As a kernel of its own it is bandwidth-bound, and its time does not depend on the batch size, so at small batch sizes it can take a large share of the step.

The update is the `multi_tensor_apply_kernel` line, about a sixth of the device time. It moves 12 bytes for each of the 34 million weights, 0.4 GB, which takes 0.4 ms at the card's bandwidth.

The four steps of the method can now be stated with their tools. **Measure** with `Benchmark` and a profiler. **Classify**: a busy device dominated by matrix products is compute-bound; a busy device dominated by elementwise and memory-bound kernels is bandwidth-bound; a device idle between kernels is overhead-bound. **Fix** what limits the program: faster formats and more devices for compute, fusion and recomputation for bandwidth, larger calls and graph capture for overhead. **Re-measure**, because each fix moves the bottleneck. We can now place any operation or program in its regime. We cannot yet say where P and β come from, or why the ridge point has risen over the hardware generations; Section 13.2 answers both questions.

13.1.5 Summary

An operation on a GPU is limited by arithmetic, by memory bandwidth, or by the fixed cost of issuing it from the host. Its arithmetic intensity, the FLOPs it performs per byte it moves, decides between the first two through the roofline bound $\min(P, I\beta)$. Below the ridge point P/β, about 164 FLOP/byte on the RTX 4090, memory sets the time; the intensity of a linear layer grows with the batch size. The third limit shows only in measurement: an operation whose device time is shorter than the host's time per call runs at the host's pace, so a 256×256 product reaches less than two percent of the peak. Correct timing waits for the asynchronous device before starting and before stopping the clock, keeps synchronization points out of loops, and averages many calls, as `d2l.Benchmark` does. The rest of the chapter applies one method, with a profiler to account for every kernel of a program: measure, classify the regime, fix it, and measure again.

13.1.6 Exercises

1. **The smallest dimension.** The exact expression for the intensity of the product $\mathbf{XW}$ in Equation 13.1, the one before the approximation, holds for all shapes, not only for $B \ll D, F$. Let $m = \min(B, D, F)$.
    1. Show that the intensity lies between $2m/(3b)$ and $2m/b$. Conclude that in bf16, on a GPU whose ridge point is 164 FLOP/byte, every matrix product with a dimension below 164 is bandwidth-bound in the roofline model.
    2. Apply the bound to three bf16 products: the final layer of the MLP of Section 13.1.4, which maps width 4096 to ten outputs, on a batch of 4096 examples; the attention scores $\mathbf{QK}^\top$ of one head with $\mathbf{Q}, \mathbf{K} \in \mathbb{R}^{T \times 64}$ for a context of T tokens; and a 1×1 convolution (Section 6.4.3) from 64 to 64 channels on a batch of 128 images of 32×32 pixels. For each product, name the dimensions that would have to grow for it to reach the ridge point, and state whether a practitioner controls them.
    3. A multi-head attention layer computes the scores of all its heads in one batched kernel. Does batching the heads change the intensity of the scores? State what your answer implies for an attention implementation that writes the $T \times T$ scores to memory.
2. code **Batch size and the ridge.** Consider the product $\mathbf{XW}$ with $\mathbf{X} \in \mathbb{R}^{B \times 8192}$ and $\mathbf{W} \in \mathbb{R}^{8192 \times 8192}$ in bf16.
    1. Using Equation 13.1 without the approximation, find the batch size at which the product reaches the ridge point of your GPU. Compare it with the batch size that the approximation gives.
    2. Predict the roofline bound on the throughput for $B \in \{1, 8, 64, 512, 4096\}$. Measure the achieved TFLOP/s with `Benchmark`, and plot the bound and the measurements against B on log-log axes. At $B = 1$, also convert the measured time into an achieved memory bandwidth, and compare it with the specification.
    3. When a language model generates text one token at a time, each of its linear layers runs a product of this kind with B equal to the number of sequences served together. What does your plot imply about the time per step of serving 64 sequences instead of one?
3. code **Hidden barriers.** Each of the following lines runs once per step of a training loop, where `x` and `g` are tensors on the GPU and `c` is a Python float: `x[x > 0].sum()`,

`print(x.shape)`, `x.argmax()`, `if g.norm() > c: g *= c / g.norm()`, and `x * torch.tensor(0.5, device=x.device)`.

1. Decide which of the lines make the host wait for the device, and justify each decision.
2. Rewrite every line that waits so that it computes the same result without a barrier.
3. Call `torch.cuda.set_sync_debug_mode('warn')` before the loop, which makes PyTorch warn at every operation that synchronizes with the device, and check both answers.

4. **Diagnosis from timings.** Suppose that `Benchmark` reports the following times per call for four operations of a bf16 program on the RTX 4090 ($P = 165$ TFLOP/s, $\beta = 1.008$ TB/s). Operation A multiplies two 8192×8192 matrices in 7.4 ms. Operation B applies a softmax to the rows of an 8192×8192 matrix in 0.30 ms; it reads and writes the matrix once and performs about five FLOPs per element. Operation C multiplies two 256×256 matrices in 10 µs. Operation D multiplies two 1024×1024 matrices in 40 µs.

   1. For each operation, compute its arithmetic intensity and the fractions of P and of β that it achieves. Classify each operation twice: once by comparing its intensity with the ridge point, and once by the roof that it is close to. On which operations do the two classifications disagree?
   2. For each operation that is close to neither roof, name the most likely of the causes given in Section 13.1.3.1, and describe what a profiler timeline of the repeated calls would show that confirms it.

5. **Idle time in a profile.** Suppose that two programs keep the GPU busy for only a fifth of their wall-clock time. Program A trains a small MLP with many narrow layers and launches every operation from Python. Program B trains a large convolutional network, and its data loader decodes and resizes every image in the main Python process.

   1. Describe where the idle gaps of each program appear in a timeline. For each program, state whether one of the fixes of the method at the end of Section 13.1.4 removes its gaps, and name that fix where one does.
   2. Describe an experiment that confirms the diagnosis of program B without a profiler.
   3. A third program keeps the GPU busy with large matrix products but converts its loss to a Python float after every step, and its profile attributes most of the host time to that conversion. Explain what the host and the device do during that time, and why that time is not overhead in the sense of the overhead-bound regime.

6. code **A pipeline model of dispatch.** ⊠ Model one call of a GPU operation by three durations: the host needs h to enqueue the call, the device needs d to execute it, and a barrier adds a latency σ after the queue drains. The device executes calls in the order in which they were enqueued, and it is idle when the first call is enqueued.

   1. Show that R calls enqueued back to back and followed by one barrier take

$$R \max(h, d) + \min(h, d) + \sigma. \tag{13.3}$$

   What does `Benchmark` therefore report for an operation with $d < h$? Explain why its result for `x + 1` on a 256×256 tensor says nothing about the memory bandwidth of the kernel.

2. Treat one step of the loops of Section 13.1.2 as one call. The loop that converts the running total into a Python number on every step waits at a barrier after each step. Express the ratio of its time to the time of the loop that reads the total once in terms of h, d, and σ.

3. Run both loops with `x` of side 256, 512, 1024, 2048, and 4096. From the ten timings, estimate h and σ, which do not depend on the size, and d for each size, and check that the model reproduces the measured ratios. Above which device time per step does reading a value on every step cost less than ten percent?

7. code **Tiles and waves.** ▣ Model a matrix-product kernel on a GPU with S SMs as follows; the RTX 4090 has $S = 128$. The kernel splits the $n \times n$ output into 128×128 tiles. Each SM computes one tile at a time at $1/S$ of the peak, and the tiles run in waves of at most S.

   1. Derive the fraction of the peak that this model allows as a function of n and S, and evaluate it with the S of your GPU for the sizes of the sweep in Section 13.1.3.1. Compare the result with the fractions of the peak that your run of the sweep measures, and name the sizes at which the host time per call, not the tiles, limits the measurement.
   2. Predict the fraction for $n = 2176$, 2304, 2432, and 2560, all multiples of 128, relative to the fraction at $n = 2048$. Measure the five sizes with `Benchmark`, and explain where the measured pattern departs from the predicted one.
   3. Describe two ways in which a kernel can split the product at $n = 512$ into more pieces of work, so that all SMs take part, and state the cost that each way adds.

   Adapted from NVIDIA, Matrix Multiplication Background User's Guide, Section 3.2, Wave Quantization.

8. [extended] **Roofline of a ResNet step.** ▣ Train `ResNet18` of Section 7.4 on Fashion-MNIST at 96×96 pixels with batch size 128, as in that section, and map one training step onto the roofline of your GPU.

   1. For every convolution of the forward pass and for the final dense layer, compute the FLOPs and the bytes, counting the input, the weights, and the output once each. Do the same for one batch normalization and one ReLU of each stage. Mark every operation at its intensity on a roofline plot of your GPU.
   2. Profile one training step. Split its device time among convolutions, normalizations and activations, and all other kernels, add the idle time between kernels, and report the share of each in the wall-clock time of the step.
   3. Apply the fix of the method that targets the largest share outside the compute-bound regime, measure again, and report which share dominates afterward.

13.2 Hardware

The roofline model of the previous section describes a GPU by two numbers from its specification sheet, its peak arithmetic rate P and its memory bandwidth β. An operation that performs I FLOPs per byte it moves runs at most at $\min(P, I\beta)$. On the RTX 4090 memory lags far behind arithmetic: the card performs up to 165 trillion floating-point operations per second on bf16 matrices, but it reads its own memory at only about one trillion bytes per second. This section asks where the two numbers come from, and why the bandwidth falls so far short of the arithmetic rate.

The gap decides the speed of a task we met in Section 11.3, generating text one token at a time. Suppose we generate with a GPT of one billion parameters. In bf16 its weights occupy 2 GB, and every generated token needs all of them. The arithmetic for one token, about 2×10^9 FLOPs, takes the tensor cores 12 µs, while reading the 2 GB of weights takes the memory 2 ms. The card therefore produces at most about 500 tokens per second for a single stream of text, and its arithmetic units sit idle more than 99% of the time. Peak arithmetic does not predict speed, because every operand must first arrive through a memory system that is far slower.

The explanation rests on three physical facts. Data must cross the boundaries between chips. The wires that cross them grow in number more slowly than the transistors that compute. And moving a byte takes far more energy than computing with it. Together they explain why the ratio P/β, the ridge point, is large and has grown over the years. The same reasoning covers the paths to the host and between GPUs, and our worked example is this book's build machine, a workstation with four RTX 4090s.

The numbers below are those of late 2026. They will change, but the relations between them change slowly: memory is orders of magnitude slower than arithmetic, each chip boundary lowers bandwidth by roughly another order of magnitude, and compute grows faster than bandwidth, which grows faster than capacity. Hennessy and Patterson (2011) treat computer architecture in depth; Section 39.4 gives purchase advice.

13.2.1 The Memory Hierarchy

Every byte a program touches lives somewhere in a hierarchy of memories (Figure 13.4), and each level trades capacity for speed. The GPU's on-die memories, its registers and caches, hold about a tenth of a gigabyte and deliver tens of terabytes per second. The GPU's own DRAM, the *high-bandwidth memory* (HBM) of datacenter parts or the GDDR of consumer cards, holds tens to hundreds of gigabytes at one to eight terabytes per second. Host DRAM holds hundreds of gigabytes at about half a terabyte per second per CPU socket. NVMe flash holds terabytes at about fourteen gigabytes per second. Network storage has no practical limit on capacity but delivers only what the network card can carry.

13.2.1.1 Bandwidth and Latency

Figure 13.5 ranks the data paths of a training job by bandwidth, from the 14 GB/s of an NVMe drive to the 8 TB/s of the HBM next to an NVIDIA B200 GPU, almost three orders of magnitude. The steps follow physical boundaries. HBM sits in the same package as the GPU die, millimeters away and connected to it by thousands of short wires. Every connector, board trace, or cable that data must cross reduces the bandwidth. As a rule of thumb, **each chip boundary lowers the bandwidth by roughly an order of magnitude**. On our build machine the two most important paths differ by a factor of thirty. The RTX 4090 reads its GDDR6X memory at 1 TB/s, while its PCIe 4.0 link to the host carries 32 GB/s in each direction.

Bandwidth says how fast data flows once it flows. *Latency* says how long the first byte takes to arrive. Figure 13.6 shows that latencies span eight orders of magnitude, from a one-nanosecond hit in a CPU's L1 cache to a sixty-millisecond round trip across a continent. The GPU cache and memory entries come from microbenchmarks of the RTX 4090 (Luo et al. 2024). The highlighted entry matters most in this chapter. A framework that dispatches an operation from Python spends 5–15 µs of host time on it, of which the CUDA driver's kernel launch takes 1–5 µs. In that time the RTX 4090 could stream 5–15 MB from its memory, more than many operations of a small model read. This host cost creates the overhead

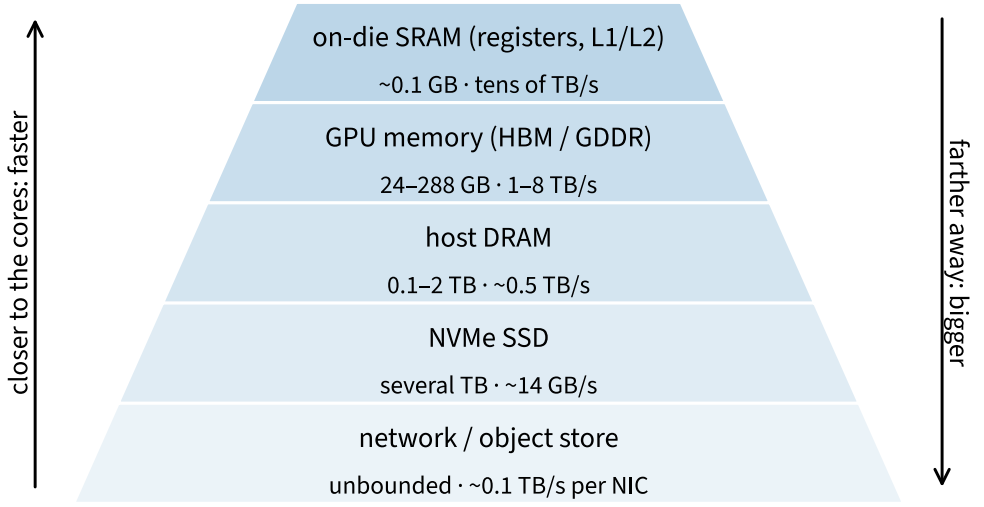


Figure 13.4: The memory hierarchy in late 2026. Each level downward holds orders of magnitude more bytes and delivers orders of magnitude fewer of them per second.

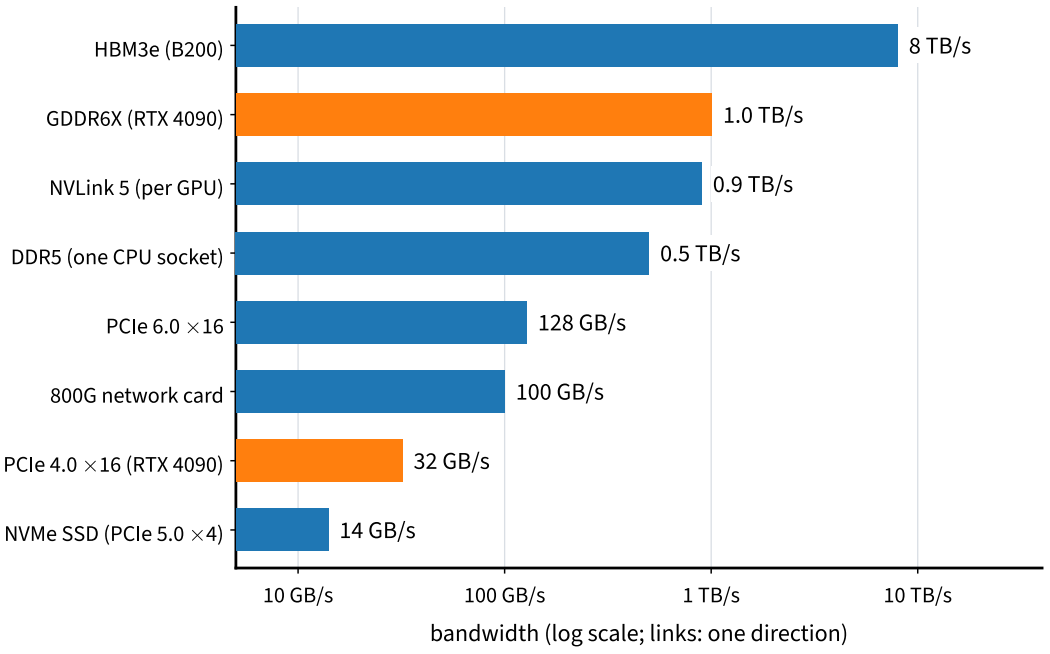


Figure 13.5: Bandwidth of the data paths that a training job uses, late 2026. Links count one direction. The build machine's own memory and host link are shown in orange.

regime of Section 13.1, and it is the reason this chapter keeps returning to fewer, larger operations.

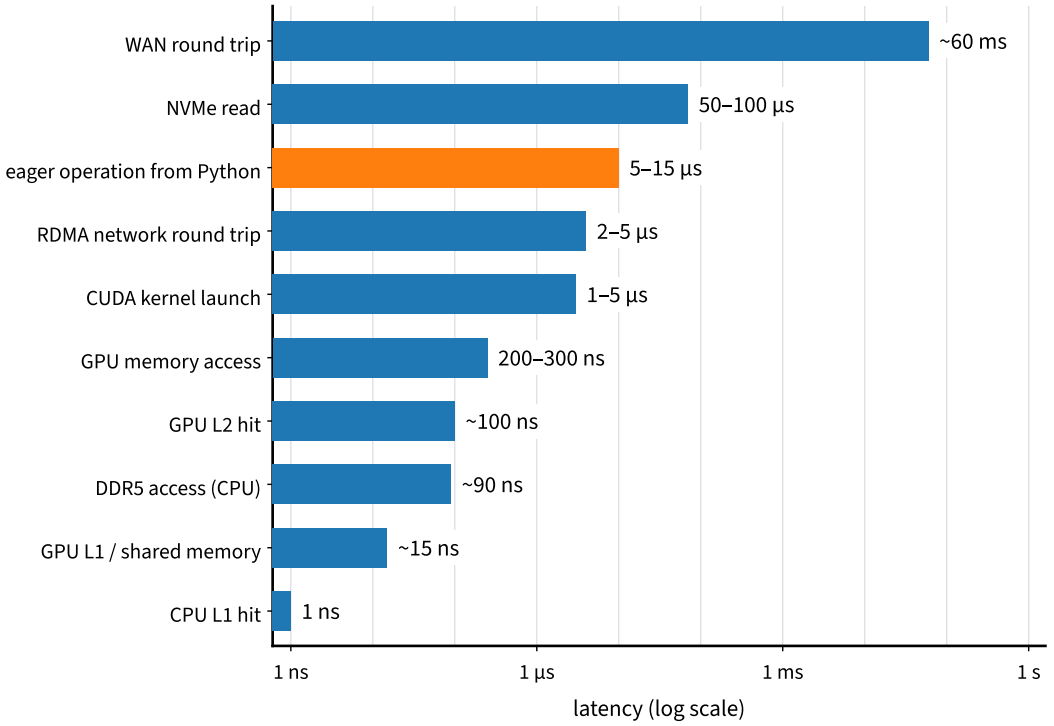


Figure 13.6: Latencies on a logarithmic scale. The highlighted entry is the host time of one framework operation dispatched from Python, of which the kernel launch is a part.

Latency and bandwidth meet in the *access pattern*. DRAM delivers its rated bandwidth only for *bursts* of consecutive addresses: opening a new row takes on the order of a hundred nanoseconds, after which the following bytes stream at full rate. A sequential read waits out the latency once per burst, a random read once for every few bytes it uses. Streaming therefore approaches the rated bandwidth, while pointer chasing and scattered gathers reach only a fraction of it. Caches help only when accesses are local and predictable. Dense tensor operations stream by design, since a matrix multiplication reads its operands in large tiles. Sparse and gather-heavy workloads do not, and a dataset that is read in shuffled order is best shuffled in large blocks.

13.2.1.2 Measured Memory Bandwidth

The number from this hierarchy that matters most for the roofline is the bandwidth that our card achieves in practice. A large elementwise operation measures it: multiplying a 1 GB vector by 2 reads and writes every element once, so we know the bytes moved, and the arithmetic is negligible.

```
x = torch.randn(256 * 1024 * 1024, device=d2l.try_gpu())  # 1 GB fp32
t = d2l.Benchmark(lambda: x * 2.0).time
```

```
print(f'measured {2 * x.nbytes / t / 1e12:.2f} TB/s '
      f'(spec: about 1.0 TB/s)')
```

```
measured 0.92 TB/s (spec: about 1.0 TB/s)
```

The kernel streams about 0.9 TB/s, within about ten percent of the specification. Every large elementwise operation in a network runs at this rate, whatever it computes, because its bytes and not its arithmetic set its time.

13.2.2 Compute and Memory Bandwidth

An operation is limited by arithmetic only if it performs at least as many FLOPs per byte as the ridge point. The larger the ridge point, the more operations are limited by bandwidth instead. The ridge point of the RTX 4090 is 164 FLOP/byte, and that of a B200 is 281 (Table 13.2 below). Both numbers follow from the geometry of chips, and the same geometry predicts that the ratio tends to rise over time.

13.2.2.1 Area and Perimeter

Arithmetic units fill the interior of a die, so compute scales with the die's area. The wires that leave the chip, through the I/O circuits that talk to memory, sit on the die's edge. Doubling the side length of a die quadruples the number of arithmetic units, but the edge only doubles: with side length n, compute grows as n^2 and I/O as n (Figure 13.7). Smaller transistors have the same effect. Halving the feature size quadruples the units on the same die, while the I/O along its edge barely grows, because the solder bumps that connect the edge to the package shrink far more slowly than transistors. Chip designers call the edge the *shoreline*. Because it grows more slowly than the interior, the number of bytes per FLOP that a chip can feed itself tends to fall from one generation to the next.

Chip designers work around the short shoreline in two ways. The first is HBM. It stacks DRAM dies vertically and places the stack on a silicon *interposer* next to the GPU die, where thousands of short, dense wires replace the few hundred long board traces of a socketed memory module. This is how a B200 reaches 8 TB/s while a CPU socket manages about 0.5 TB/s. The DRAM cells are no faster; there are many more wires, and they are much shorter. The second is to put several dies into one package. Lithography caps the size of a die at the *reticle limit*, about $858\ \text{mm}^2$, and flagship accelerators are built at that limit. A B200 consists of two such dies, joined by a bridge on the package, and acts as one GPU. Beyond one package, GPUs must exchange data over links, whose bandwidth Section 13.2.4.2 examines.

13.2.2.2 Generational Trends

Across NVIDIA's flagship datacenter GPUs, from the V100 of 2017 through the A100 and H100 to the B200 of 2024, dense 16-bit throughput grew about 2.6-fold per generation, memory bandwidth about 2.0-fold, and memory capacity about 1.8-fold. A study of twenty years of server hardware finds the same order: peak compute grew 3.0-fold every two years, DRAM bandwidth 1.6-fold, and interconnect bandwidth 1.4-fold (Gholami et al. 2024). Because compute outgrows bandwidth, the ridge point rises: from about 140 FLOP/byte on the V100 to about 290 on the H100. Workloads that were compute-bound a decade ago are bandwidth-bound today.

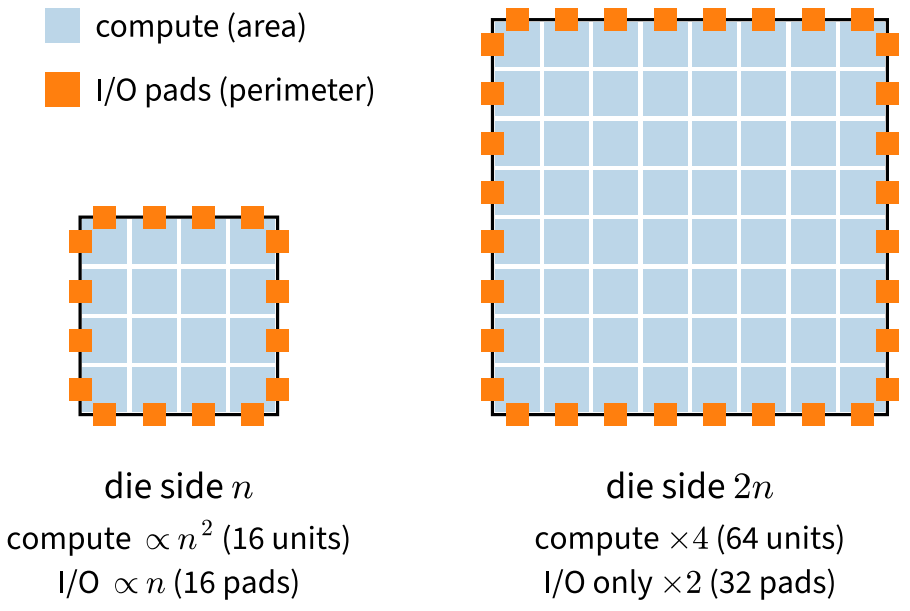


Figure 13.7: Compute is area, I/O is perimeter. Doubling the side of a die quadruples its compute but only doubles the room for I/O.

A single generation need not follow the trend, because which roof grows faster from one product to the next is a design decision. From the H100 to the B200, compute grew 2.3-fold and bandwidth 2.4-fold, so the ridge point fell slightly, from 295 to 281 FLOP/byte. From the RTX 4090 to the RTX 5090, bandwidth grew much faster than compute, and the ridge point fell from 164 to 117 (Table 13.2). Over many generations, compute still grows faster than bandwidth, and the ridge point rises.

13.2.2.3 Energy per Operation

Energy gives a second reason why data movement dominates. Figure 13.8 shows the energy of elementary operations in a 45 nm process (Horowitz 2014). An 8-bit integer addition costs 0.03 pJ and an fp32 multiplication 3.7 pJ, while reading 64 bits from DRAM costs 1.3–2.6 nJ, about as much as 500 fp32 multiplications. Reads from on-chip SRAM lie in between, and they cost more the larger the array, from 10 pJ for 8 KB to 100 pJ for 1 MB, because the bits travel farther. Newer processes lower every entry, but the order stays the same: moving data costs more than computing with it, and the farther it moves, the more it costs.

The techniques of this chapter save energy for the same reason that they save time. Fusion (Section 13.3) keeps intermediate results on the die instead of writing them to DRAM. Narrower formats (Section 13.4) halve the bytes of every operand. Recomputation uses cheap arithmetic to avoid expensive memory traffic. Each performs more arithmetic per byte fetched.

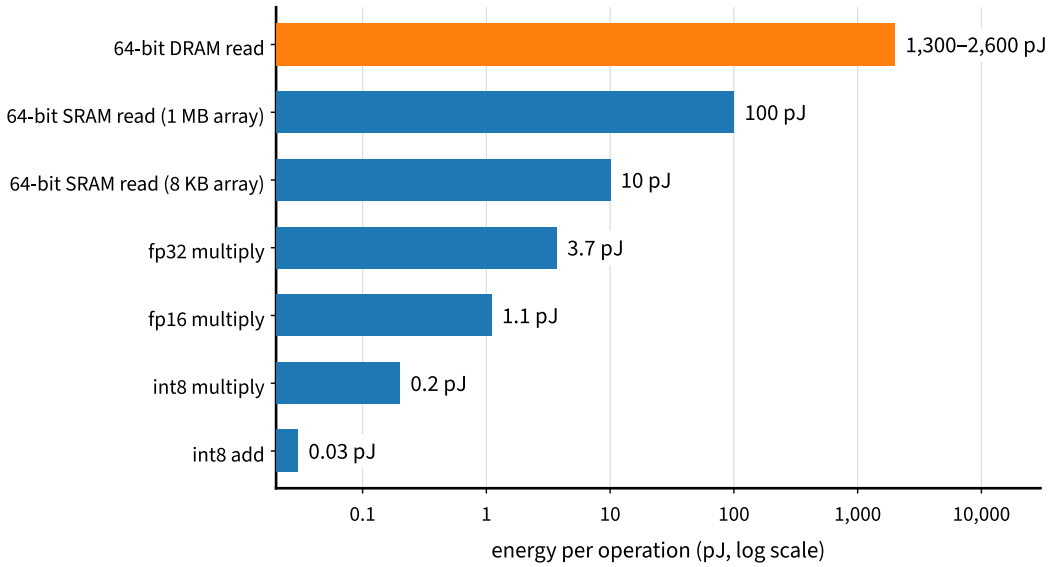


Figure 13.8: Energy per operation in a 45 nm process. One 64-bit DRAM read costs about as much as 500 fp32 multiplications.

13.2.3 The GPU

The two roofline numbers of a GPU come from two kinds of units: memory interfaces on the shoreline, which set β, and arithmetic units in the interior, which set P. The arithmetic side also decides which number formats run fast.

13.2.3.1 Streaming Multiprocessors

A GPU spends its area on thousands of simple arithmetic units rather than on the elaborate control logic of a CPU core. The units are grouped into *streaming multiprocessors* (SMs); the RTX 4090 has 128 of them. Each SM runs threads in groups of 32, called *warps*, that execute in lockstep, and it switches among many resident warps to hide memory latency: while one warp waits a few hundred nanoseconds for its data, others compute. That is as much microarchitecture as this book needs. When a profiler reports *occupancy*, it means the fraction of an SM's warp slots that hold resident warps, which are available to hide latency. A kernel that is too small for the machine has too few warps to keep 128 SMs busy, and launch cost and latency, rather than bandwidth or arithmetic, set its time.

13.2.3.2 Tensor Cores and Number Formats

What makes GPUs deep learning machines is a further specialization, the *tensor core*: a unit that multiplies two small matrix tiles and accumulates the product in a single instruction. A scalar unit spends most of its energy on fetching and decoding an instruction for every multiplication. A tensor core spreads that cost over a whole tile, the same idea as the systolic array of Google's TPU (Jouppi et al. 2017; Kung 1988). On an H100, bf16 matrix arithmetic runs at 989 TFLOP/s and ordinary fp32 arithmetic at 67 TFLOP/s, a factor of fifteen. Consumer cards devote less of their area to tensor cores; on the RTX 4090 the factor is two in bf16 and four in fp8. Either way, matrix multiplications are the fastest operations on the

chip, provided their operands come in a format that the tensor cores support.

Section 5.5 introduced these formats: a sign bit, exponent bits that set the *range*, and mantissa bits that set the *precision*. fp32, tf32, and bf16 share an 8-bit exponent and therefore the same range, while fp16 trades range for precision and needs loss scaling to keep small gradients from underflowing. Figure 13.9 draws the formats that current tensor cores accept, down to 4 bits. Two consequences of the width matter for the roofline.

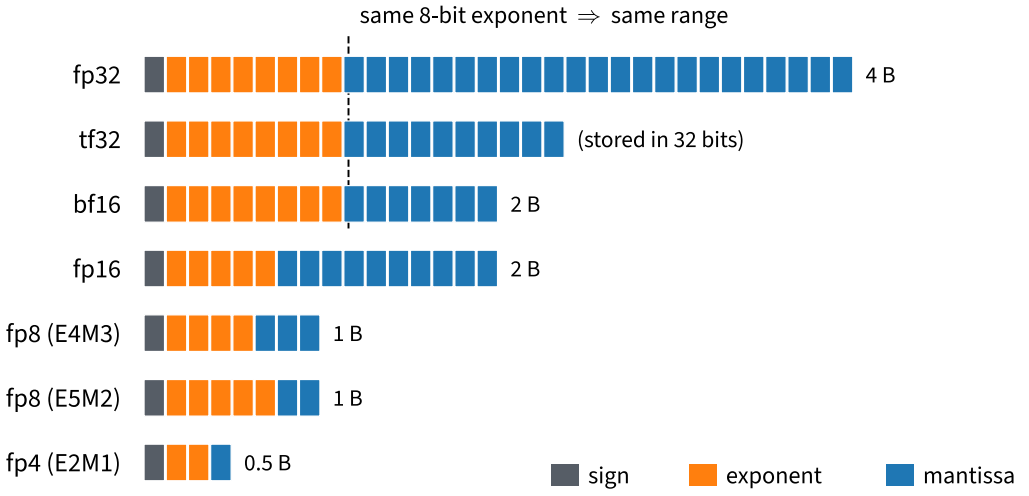


Figure 13.9: Floating-point formats drawn bit for bit. fp32, tf32, and bf16 share an 8-bit exponent and hence a range. tf32 is a compute mode of the tensor cores whose values are stored in 32 bits; every other step down halves the bytes per value.

First, **a narrower format moves fewer bytes**. Halving the width halves the bytes of every operand, so a bandwidth-bound operation runs up to twice as fast, and the weights of a model take half the memory. tf32 is the exception. The tensor cores compute with its 10-bit mantissa, but its values are stored in 32 bits, so it raises the compute roof without saving any memory traffic.

Second, **a narrower format usually raises the compute roof**. A multiplier for 8-bit values takes much less area than one for 16-bit values. On most current parts each halving of the width therefore roughly doubles the peak FLOP/s: the RTX 4090 performs 165 TFLOP/s in bf16 and 330 TFLOP/s in fp8. The ratio is a design choice, though. Intel's Gaudi 3 runs fp8 at its bf16 rate, and NVIDIA's B300 runs fp4 at three times its fp8 rate.

Formats of 8 bits and fewer cover so little range that their values are stored together with a scale factor. fp8 comes in two variants, E4M3 for precision and E5M2 for range (Micikevicius et al. 2022), and one scale per tensor works for it in most layers. That is the usual recipe on the RTX 4090 and the H100, whose tensor cores have no support for finer scales. At 4 bits one scale per tensor fails. The sixteen bit patterns of fp4 (E2M1) encode only fifteen distinct numbers, from -6 to 6, so a scale set by a few large outliers rounds most other values of the tensor to zero. Blackwell-generation NVIDIA and AMD CDNA 4 tensor cores therefore scale small blocks of values separately. In the *microscaling* formats every block of 32 values shares one power-of-two scale (Rouhani et al. 2023), and NVIDIA's NVFP4 format shares a

finer scale among 16 values. The recipes that train large models in these formats belong to the Language Models part.

Training and inference use the formats differently. Training keeps master weights and optimizer state in fp32 and accumulates matrix products in fp32, with bf16 as the usual format for the arithmetic (Section 13.4). Inference runs forward only and can use the narrowest format that preserves the quality of the model.

13.2.3.3 Accelerator Specifications

Table 13.2 collects the two roofline numbers, and the other quantities of this section, for two datacenter GPUs and two consumer cards, including our own.

Table 13.2: Representative GPUs in late 2026, from vendor datasheets (B200: per GPU of NVIDIA's DGX B200 system). Throughput is dense (without 2:4 sparsity) with fp32 accumulation at boost clock, in TFLOP/s (TF); links count one direction.

	H100 SXM	B200	RTX 4090 (ours)	RTX 5090
memory	80 GB HBM3	180 GB HBM3e	24 GB GDDR6X	32 GB GDDR7
memory bandwidth	3.35 TB/s	8 TB/s	1.0 TB/s	1.8 TB/s
bf16	989 TF	2,250 TF	165 TF	210 TF
fp8	1,979 TF	4,500 TF	330 TF	419 TF
fp4	—	9,000 TF	—	1,676 TF
ridge point (bf16)	295 FLOP/B	281 FLOP/B	164 FLOP/B	117 FLOP/B
link to other GPUs	NVLink 4, 450 GB/s	NVLink 5, 900 GB/s	PCIe 4.0 ×16 via host memory, 32 GB/s	PCIe 5.0 ×16 via host memory, 64 GB/s
power	700 W	1,000 W	450 W	575 W

The RTX 5090's fp4 entry is four times its fp8 entry, not two times, because GeForce cards run fp8 and fp16 tensor arithmetic at half rate when they accumulate in fp32, and fp4 carries no such penalty. Accelerators from other vendors, such as AMD's Instinct GPUs, Google's TPUs, and AWS's Trainium chips, follow the same design of stacked memory beside a die full of matrix units, and the roofline reasoning applies to them unchanged.

Capacity, the slowest-growing of the three quantities, decides what fits at all. Training our 1B-parameter GPT with Adam in fp32 stores 16 bytes per parameter for the weights, their gradients, and the two moment estimates, 16 GB in all before any activations (Section 13.4 breaks this total down). An RTX 4090 holds that with 8 GB to spare for activations; a model twice as large does not fit.

13.2.4 Data Paths outside the GPU

A GPU computes only on data in its own memory. Everything else arrives over links that are one to two orders of magnitude slower: from the host, which holds the dataset, and from other GPUs, which hold the other replicas of a model.

13.2.4.1 The Host

The CPU no longer does the arithmetic of a deep learning machine: one RTX 4090 delivers more than ten times the fp32 arithmetic of a 64-core server CPU. It still does three jobs that can limit training. First, the CPU orchestrates. A CPU thread launches every kernel the GPU runs, at 5–15 µs of host time per eager operation (Figure 13.6), until the compilation of Section 13.3 takes Python out of the loop. Second, the CPU runs the input pipeline: decoding images, tokenizing text, augmenting and batching. If the pipeline falls behind, the GPU waits, whatever its peak throughput.

Third, the CPU feeds the bus, since every host-to-device copy crosses PCIe. As Section 5.7 showed, an ordinary, *pageable* host array is first copied into page-locked memory, which the GPU's copy engine can read directly, and only then transferred. An array that already sits in pinned (page-locked) memory skips the first copy, and its transfer can overlap with computation. We time both kinds of copy for a 256 MB array to find the ceiling of the host link.

```
x_pageable = torch.randn(64 * 1024 * 1024)  # 256 MB on the host
x_pinned = x_pageable.pin_memory()
gpu = d2l.try_gpu()
for desc, src in [('pageable', x_pageable), ('pinned', x_pinned)]:
    t = d2l.Benchmark(lambda: src.to(gpu, non_blocking=True)).time
    print(f'{desc}: {src.nbytes / t / 1e9:.1f} GB/s')
```

```
pageable: 15.0 GB/s
pinned: 24.2 GB/s
```

`pin_memory()` returns a copy of a tensor in page-locked memory, and `non_blocking=True` issues the copy to the GPU from pinned memory without waiting for it to finish. From pageable memory the copy is synchronous, whatever the flag says. A `DataLoader(pin_memory=True)` pins every minibatch on a background thread of the main process as the workers deliver it, so that the training loop can issue non-blocking copies.

The pinned copy reaches about 23 GB/s, close to the 32 GB/s that a PCIe 4.0 ×16 link carries in each direction and forty times below the card's own memory bandwidth. The pageable copy is slower, because every byte first makes an extra trip through host memory. Data that the GPU reads repeatedly therefore belongs on the device. Data that must arrive at every step, such as minibatches, should come from pinned memory, with the copy overlapped with computation.

13.2.4.2 Links between GPUs

When one GPU is not enough, GPUs exchange gradients, parameters, or activations over links that are much slower than their memories. Datacenter GPUs have a dedicated fabric for this. NVLink 5 gives each B200 1.8 TB/s to its peers, 900 GB/s in each direction, about a tenth of its memory bandwidth. A GB200 NVL72 rack joins 72 GPUs into one NVLink domain. Consumer GPUs have only PCIe; the RTX 3090 and 3090 Ti were the last GeForce cards with an NVLink connector. NVIDIA also does not support peer-to-peer transfers between RTX 4090 or RTX 5090 cards, so two such cards in one machine cannot send data to each other directly even over PCIe.

Our build machine shows the consequence (Figure 13.10). Its four RTX 4090s sit in pairs behind two PCIe host bridges. Without peer-to-peer access, every byte from one GPU to

another is staged through host memory: up one PCIe link, through the CPU's memory, and down another. A copy between two of our GPUs therefore runs at most at the rate of one PCIe 4.0 link, the 23 GB/s that the pinned copy above reached, about 40 times less than one direction of NVLink 5. Collective communication libraries such as NCCL reach even less on this staged path at their default settings, and Section 13.6 traces where their time goes.

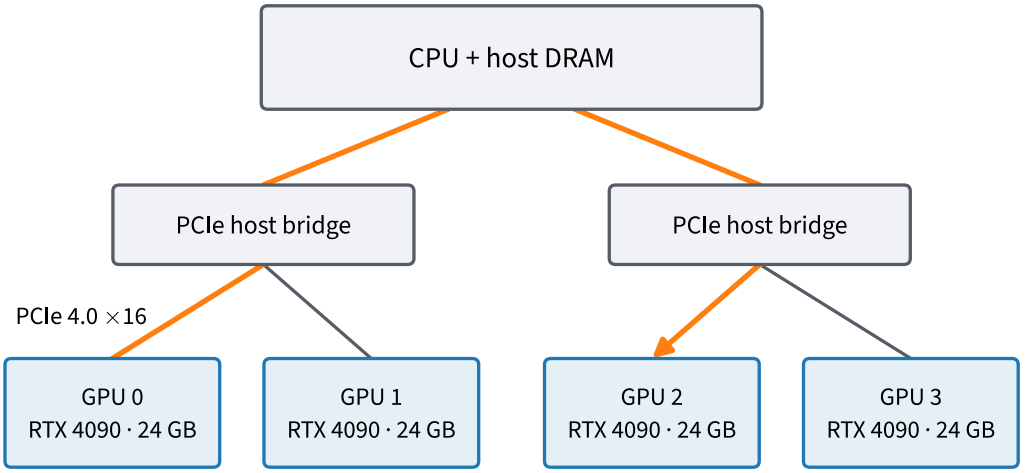


Figure 13.10: This book's build machine, as reported by `nvidia-smi topo`: four RTX 4090s in pairs behind two PCIe host bridges, with neither peer-to-peer access nor NVLink. The highlighted path is the route of every byte from one GPU to another.

Data-parallel training of our 1B-parameter GPT on two GPUs sums the 4 GB of fp32 gradients of every step across the GPUs. Moving 4 GB once takes at least 0.17 s at 23 GB/s and less than 5 ms over one direction of NVLink 5. On our machine, communication therefore adds at least 0.17 s to every step of this model, unless it overlaps with computation. Section 13.5 turns such numbers into a cost model and checks its predictions on this machine.

13.2.5 Workloads on the Roofline

The numbers of this section place a single language model in two regimes, as Section 11.3 measured on this card. *Prefill* pushes a prompt of n tokens through the network at once and uses every weight fetched from memory for all n tokens, an intensity of about n FLOPs per byte. A prompt of a thousand tokens lies far above any ridge point. *Decode* generates one token per step. For a GPT with N parameters it performs about $2N$ FLOPs and reads every weight and the key–value cache, an intensity of about one FLOP per byte (Section 11.3.2.1). Prefill runs at the speed of the tensor cores, decode at the speed of memory.

13.2.5.1 Single-Stream Decoding

Since a decode step is bandwidth-bound, the bytes it reads limit the rate at which a single stream of text grows. With weights in bf16, $2N$ bytes,

$$\text{tokens/s} \lesssim \frac{\beta}{\text{bytes per token}} = \frac{\beta}{2N + \text{KV bytes}}. \tag{13.4}$$

For our 1B-parameter GPT, 2 GB of weights at the 0.9 TB/s we measured cap decode near 450 tokens per second on one RTX 4090. Faster arithmetic does not raise this bound. Storing the weights in a narrower format does, in proportion to the bytes it saves. Prefill gains instead from every format that raises the compute roof.

13.2.5.2 Batched Decoding

Batching moves decode up the roofline. A step for B sequences reads the weights once and produces B tokens, while every sequence still reads its own key–value cache (Section 11.3.2.2). The measurement of Section 11.3 stopped at 64 sequences, where the step is still bandwidth-bound. Here we follow the weights alone past the ridge point. We time one bf16 weight matrix of a large model, $16{,}384 \times 16{,}384$ (0.54 GB), multiplied by a batch of B activation vectors, one row per sequence, for B from 1 to 1,024.

```
d = 16384
W = torch.randn(d, d, dtype=torch.bfloat16, device=d2l.try_gpu())  # 0.54 GB
for B in (1, 4, 16, 64, 256, 1024):
    X = torch.randn(B, d, dtype=torch.bfloat16, device=W.device)
    t = d2l.Benchmark(lambda: X @ W).time
    print(f'B = {B:4d}: {1e3 * t:5.2f} ms, {B / t:9,.0f} rows/s, '
          f'{2 * B * d * d / t / 1e12:5.1f} TFLOP/s')
```

```
B =    1:  0.57 ms,     1,759 rows/s,   0.9 TFLOP/s
B =    4:  0.57 ms,     6,996 rows/s,   3.8 TFLOP/s
B =   16:  0.57 ms,    27,948 rows/s,  15.0 TFLOP/s
B =   64:  0.58 ms,   109,610 rows/s,  58.8 TFLOP/s
B =  256:  0.86 ms,   297,997 rows/s, 160.0 TFLOP/s
B = 1024:  3.29 ms,   311,400 rows/s, 167.2 TFLOP/s
```

The product performs $2Bd^2$ FLOPs on $2d^2$ bytes of weights, an intensity of about B FLOPs per byte. For $B = 1$ it is bandwidth-bound and takes about 0.6 ms, the time to read 0.54 GB. Up to a batch of 64 the time barely changes, so the rows per second through this matrix grow almost in proportion to the batch. The roofline puts the crossover where the intensity reaches the ridge point, at a batch of about 164. Beyond it the time grows in proportion to the batch, and the arithmetic rate levels off near the 165 TFLOP/s peak. One matrix on one card covers both ends of Figure 13.1.

In a whole model the key–value caches add bytes that batching does not share, so the crossover moves to larger batches, and for long contexts it never comes. Inference servers therefore batch the requests of many users and keep the caches small. The systems that do so, with their schedulers, paged caches, and speculative decoding, belong to the Language Models part.

13.2.5.3 Rules of Thumb

With the magnitudes of Table 13.3, a workload can be placed on the roofline before it runs: count its FLOPs and bytes, divide by P and β, and compare the result with the host cost of its operations. Real programs usually fall short of both roofs, because eager execution sends every intermediate result through memory and incurs the host cost of every operation. Section 13.3 removes both costs.

Table 13.3: Rules of thumb: characteristic magnitudes and why each matters.

quantity	magnitude	why it matters
eager operation from Python	5–15 µs of host time (kernel launch 1–5 µs)	small operations cannot keep the GPU busy
GPU memory bandwidth	1–8 TB/s	the speed limit of elementwise operations and of decode
PCIe ×16, one direction	32 GB/s (4.0) to 64 GB/s (5.0)	host↔device; keep data on the device
NVLink, per GPU and direction	450–900 GB/s	datacenter GPUs exchange data an order of magnitude faster than PCIe
bytes per parameter, training with Adam	16 (fp32), 18 (mixed precision)	what fits (Section 13.4)
bytes per token, decode	$2N$ + KV cache (bf16)	tokens/s $\lesssim$ bandwidth / bytes per token
DRAM read against fp32 multiply	about 500 times the energy	fuse, narrow the format, recompute

13.2.6 Summary

Memory is a hierarchy in which each level farther from the arithmetic holds orders of magnitude more bytes and delivers orders of magnitude fewer per second. Each chip boundary lowers the bandwidth by about an order of magnitude, and latencies span eight orders of magnitude. Our RTX 4090 streams its memory at about 0.9 TB/s, close to its specification. Compute grows with the area of a die and I/O with its perimeter, and moving a byte takes more energy than computing with it. Compute has therefore outgrown bandwidth, and the ridge point of flagship GPUs has roughly doubled over four generations, to about 290 FLOP/byte, although a single generation can lower it. Tensor cores make matrix arithmetic the fastest work on a GPU in the formats they support. Halving the width of a format halves the bytes moved and, on most parts, doubles the peak FLOP/s; tf32 raises only the compute roof. Outside the GPU, bandwidth drops by one to two orders of magnitude. On our machine a copy from pinned host memory reaches about 23 GB/s over PCIe 4.0, and a copy between two GPUs, staged through host memory, runs no faster, while NVLink 5 carries 900 GB/s per direction. Prefill is compute-bound, and single-stream decode is bandwidth-bound at about β divided by the bytes per token, roughly 450 tokens per second for a 1B-parameter GPT on an RTX 4090. Batching raises the intensity of decode in proportion to the batch until it reaches the ridge point.

13.2.7 Exercises

1. code **Latency and bandwidth, measured.** A transfer of n bytes over a path with latency α and bandwidth β takes about

$$t(n) = \alpha + \frac{n}{\beta},$$

so it reaches half the bandwidth at $n_{1/2} = \alpha\beta$ (Hockney 1994). The model describes

a call that is timed to completion on its own, so every measurement below waits for the device after each call.

1. Predict $n_{1/2}$ for the elementwise kernel `x * 2.0` of Section 13.2.1, counting the bytes it reads and writes, with α the host time of an eager operation in Figure 13.6 and β the bandwidth that the code of that subsection measures on your GPU.
2. Time `x * 2.0` for tensors from 1 MB to 2 GB, and plot the achieved bandwidth against the size on logarithmic axes. Identify the regimes that the curve passes through, and relate the sizes at which it changes regime to Figure 13.6 and to the on-die capacity in Figure 13.4. In which regime does the curve depart from $t(n)$, and why?
3. Repeat the host-to-device measurement of Section 13.2.4.1 for transfers from 4 KB to 256 MB, and fit α and β. Where does the fitted α fall on Figure 13.6?

2. **Shoreline countermeasures.** Section 13.2.2 names two countermeasures to the shoreline problem: wider memory interfaces and more dies per package.

1. A DDR5-6400 DIMM has 64 data wires, each carrying 6.4 Gb/s. An HBM3e stack has a data interface of 1,024 wires, and a B200 carries eight stacks. Compute the bandwidth of one DIMM, the number of DIMMs that supply the 0.5 TB/s of a CPU socket, and the rate per wire that the B200's 8 TB/s in Table 13.2 implies. Which factor accounts for the gap between the socket and the B200, the number of wires or the rate per wire?
2. A B200 consists of two dies at the reticle limit of $858\,\mathrm{mm}^2$. Treat every die as a square. Compare the total edge length of the two dies, after setting aside one edge of each for the bridge that joins them, with the edge length of a hypothetical single die of twice the area. Then show that n dies of a fixed area offer an edge length per unit area that does not depend on n.
3. The two dies of a B200 act as one GPU. Which traffic does the split move across a chip boundary that a single die would keep on-die? What bandwidth must the bridge carry, compared with the 8 TB/s of HBM? Use the physical reason behind the chip-boundary rule of Section 13.2.1 to explain how an on-package bridge can meet this requirement.

3. **Format halving on the roofline.** Model the step from one format to a format of half its width, as Section 13.2.3 describes, by a doubling of the peak arithmetic P and a halving of the bytes that every operation moves.

1. Use the roofline bound $\min(P, I\beta)$ to compute the speedup of an operation of intensity I, and show that it does not depend on I.
2. A skinny matrix product with batch size B, much smaller than the matrix, and b bytes per element has an intensity of about $2B/b$ FLOPs per byte. Using Table 13.2, find the batch size at which this product becomes compute-bound on the B200 in bf16, fp8, and fp4, and on the RTX 5090 in the same three formats. Compare the bf16 value for the RTX 4090 with the batch sweep of Section 13.2.5.
3. Use the result of the first sub-problem to explain the pattern in the batch sizes of the previous sub-problem, and identify the entry of Table 13.2 that breaks the pattern. What does that format step give a bandwidth-bound operation, and what does it give a compute-bound one?

4. **Starved input pipeline.** The compute of a training job alone would process 2 GB/s of decoded images. One worker process prepares batches by running two stages in sequence. It reads compressed JPEG files from NVMe in shuffled order at 1 GB/s of

compressed data, and it decodes them on one CPU core at 0.3 GB/s of decoded output. Decoding expands the data eightfold. The training loop then copies each decoded batch from pageable host memory to the GPU at 12 GB/s before it computes on the batch.

1. At what rate, in decoded bytes, does one worker deliver batches? Which stage binds, and what would a drive with twice the read rate change?
2. Workers run in parallel on separate cores. How many workers keep the GPU fed?
3. With enough workers, the synchronous copy still lengthens every training step. By what fraction does it lengthen the step, and by what fraction after a switch to pinned memory at twice the pageable rate? What does an asynchronous copy change?

5. **Ridge point of a link.** The ridge point P/β of Section 13.1.1 applies to any data path, with β the bandwidth of that path: it counts the FLOPs that a device performs in the time one byte crosses the path.

   1. Compute P/β in bf16 for an RTX 4090 against its own memory and against the host-staged GPU-to-GPU path of Section 13.2.4.2 at 23 GB/s. Compute it for a B200 against its HBM and against one direction of NVLink 5, 900 GB/s.
   2. In data-parallel training (Section 13.5), every GPU holds a copy of a model with N parameters and processes T tokens per step at about $6N$ FLOPs per token. The GPUs then sum their bf16 gradients with an *allreduce*, a collective operation that sends about twice the gradient bytes from every GPU. Assume that compute runs at peak and does not overlap communication. Show that communication takes no longer than compute if and only if T exceeds a threshold proportional to the ridge point of the link, and that the threshold does not depend on N.
   3. Evaluate the threshold for the host-staged path, for NVLink, and for an effective 2 GB/s, the rate that NCCL reaches on the staged path of Figure 13.10 at its default settings. By what factor does the ridge point of each of these links exceed the memory ridge point of the same GPU? Relate the factors to the chip-boundary rule of Section 13.2.1.

6. **Energy per byte.** Use the energies of Figure 13.8, and charge every FLOP the energy of one fp32 multiply.

   1. Find the arithmetic intensity at which the energy of the arithmetic equals the energy of reading its operands from DRAM, and compare it with the ridge point of the RTX 4090. What limits the time of an operation with an intensity of 100 FLOP/byte, and what dominates its energy?
   2. A B200 is rated at 1,000 W and streams 8 TB/s from HBM. Compute the power that this traffic would draw at the DRAM energy of Figure 13.8, and the largest energy per byte that the power rating allows. Explain the discrepancy with the packaging of Section 13.2.2.
   3. An 8-billion-parameter model in bf16 decodes on a B200 at the bound of Equation 13.4, ignoring the KV cache, while the card draws its full 1,000 W. Compute the energy per generated token and the electricity cost of one million tokens at 0.30 US dollars per kWh. Repeat for 64 streams that share every weight read, after checking that the batched step is still bandwidth-bound.

7. **Memory sweep time.** The *sweep time* C/β of an accelerator with memory capacity C and bandwidth β is the time it takes to read every byte of its memory once.

1. Compute the sweep time of each GPU in Table 13.2.
2. A model in bf16 fills a fraction f of the memory with its weights. Ignore the KV cache, and use Equation 13.4 to bound the single-stream decode rate in terms of f and the sweep time. Which rate does the bound allow for the largest model that each GPU can hold?
3. Under the long-run growth rates of Section 13.2.2, by what factor does the sweep time change per hardware generation? What does this factor imply for the decode rate of the largest model that fits on each new generation?

8. **Launch-bound decode.** In eager single-stream decode, the CPU launches every kernel of every block for every token, while the GPU reads the weights of every block. Consider a GPT whose L blocks of width d each hold about $12d^2$ weights in bf16 and launch $K = 12$ kernels per token. Take 8 µs of host time per kernel, within the range that Figure 13.6 gives for an eager operation. Launches are asynchronous, so a decode step takes at least the larger of the total launch time and the time to read the weights. Ignore the embedding table and the KV cache.

    1. Show that which of the two times is larger does not depend on L, and find the width d^* at which they are equal on the RTX 4090 and on the B200.
    2. Evaluate both bounds on the decode rate of the GPT-2 configuration of Section 11.2.4, with $d = 768$ and 12 blocks, on the RTX 4090. Which bound binds?
    3. What does the change of d^* from the RTX 4090 to the B200 imply for the benefit of the capture-and-replay of Section 13.3 on faster GPUs?

9. code **Burst versus random access.** ▯ Let `x` be the 1 GB vector of Section 13.2.1, and gather it through an index vector `idx` of the same length, as `x[idx]`.

    1. Time the gather with `idx` a random permutation and with `idx` the identity permutation, which request the same elements in a different order. Report the achieved bandwidth of each, counting the index, the gathered elements, and the output, and the ratio of the two.
    2. Compare the ratio with the cost that the burst model of Section 13.2.1 assigns to a random read. Explain the difference with the resident warps of Section 13.2.3.
    3. Permute blocks of k consecutive elements instead of single elements, for k from 1 to 1,024, and plot the bandwidth against the block size in bytes. At what block size does the shuffled gather approach the identity gather? What does this size imply for the granularity of shuffled reads from a dataset?

10. **Latency hiding.** ▯ Little's law (Little 1961) states that in a steady state the average number of requests in flight equals the throughput times the average latency of one request. For a memory system, the bytes in flight therefore equal the bandwidth times the latency.

    1. Take 250 ns, the GPU memory latency of Figure 13.6, as the latency of a load on the RTX 4090, with its 1.0 TB/s and 128 SMs. How many bytes must be in flight to sustain the rated bandwidth? If every thread has one 4-byte load outstanding, how many warps per SM must be waiting on memory at any moment?
    2. Repeat the byte count for the B200 at the same latency. Bandwidth grows about $2\times$ per generation, while DRAM latency has stayed nearly constant for decades. What does this imply for the parallelism that a kernel must expose on each new generation?
    3. In pointer chasing, the address of every load comes from the previous load, so a chain of loads has one request in flight. A CPU has the DDR5 access latency of

Figure 13.6, 90 ns, and fetches 64-byte cache lines. Compute the bandwidth of one chain and the number of independent chains that would saturate a socket with 0.5 TB/s.

13.3 Compute Graphs and Compilation

In Section 13.1 the activation function `gelu_ish` took eight to nine times as long as a single addition on the same tensor, although its arithmetic hardly mattered. The cause was the way the framework ran it. Each of its eight elementwise operations ran as a GPU kernel of its own, which read its inputs from GPU memory and wrote its result back, and Section 13.2 showed why that memory traffic is slow. Can a framework run such a chain without the traffic, and without the cost of issuing every operation from Python?

A count of the bytes shows how much is wasted. The function computes $0.5\,x\,(1+\tanh\left(0.8\,(x+0.04\,x^3)\right))$. Applied to a 4000×4000 matrix in fp32, which takes 64 MB, the eight kernels together move 1.15 GB to produce 64 MB of output. Reading the input once and writing the result once would move 128 MB, nine times less. At the RTX 4090's memory bandwidth of about 1 TB/s, the eight kernels need at least 1.15 ms where 0.13 ms would do.

Activation functions, normalization layers, residual additions, and optimizer updates are all chains of this kind, and a deep network contains hundreds of them. Small tensors add a second cost: launching a kernel from Python takes longer than running it, and the GPU waits. Both costs come from *eager execution*, which runs a program one operation at a time, as Python reaches it. The remedy is to give the whole program to a compiler before it runs. A compiler that sees all eight operations can merge them into one kernel that reads x once and writes the result once. A sequence of kernels, once recorded, can be replayed without Python.

We study how a framework captures a program as a graph, what a compiler does with that graph, and which programs escape capture. Compilation addresses two of the three regimes of Section 13.1, the bandwidth-bound and the overhead-bound regime. Memory capacity is a separate limit, the subject of Section 13.4.

```
torch.set_float32_matmul_precision('high')  # fp32 matmuls may use tf32
```

13.3.1 Graph Capture

A compiler must see a program whole before it can rewrite it, but Python executes a program one line at a time. A framework therefore has to obtain the program as a graph first, and some Python constructs do not fit into a graph.

13.3.1.1 Computation Graphs

The computation of a neural network forms a *computation graph*: its nodes are operations, and its edges carry tensors from one operation to the next. Autograd records this graph during the forward pass so that it can walk it backward to compute gradients (Section 1.5). Figure 13.11 shows the graph of a two-layer network $\mathrm{relu}(\mathbf{X}\mathbf{W}_1+\mathbf{b}_1)\mathbf{W}_2+\mathbf{b}_2$ followed by a loss.

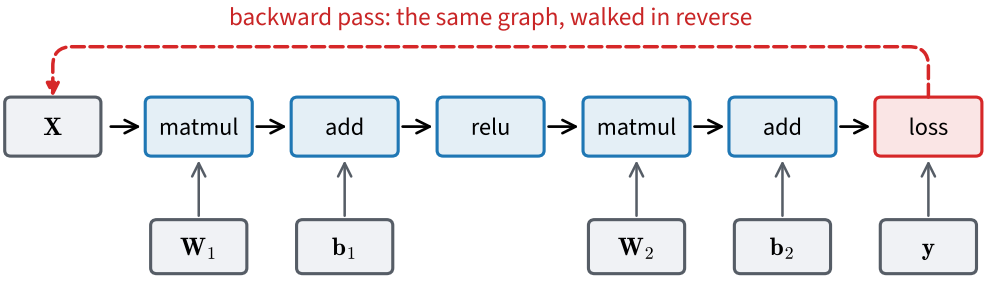


Figure 13.11: The computation graph of a two-layer network with a loss. Autograd records it to run the backward pass; a compiler that has the whole graph before the forward pass can rewrite it.

Autograd's record cannot speed up the forward pass. It is written while the operations execute, so by the time it exists every operation has already run as its own kernel. Each kernel needs a launch, which takes 1–5 µs in the CUDA driver and about 5–15 µs of host time when Python dispatches the operation (Section 13.2), and each elementwise kernel makes a round trip through GPU memory. To merge the `add` with the `relu` that follows it, a compiler needs the graph before the forward pass runs. Obtaining the graph in advance is called *graph capture*.

13.3.1.2 Tracing

The usual way to capture a graph is to run the function once on *placeholders*, objects that carry the shape and dtype of a tensor but no values. Every tensor operation applied to a placeholder is recorded instead of computed, and the record is the graph. Python code that does not depend on tensor values runs as usual and leaves only its effect: a loop over four layers leaves four copies of the layer's operations, and arithmetic on shapes leaves constants. Running a function to record its operations is called *tracing*. A compiler then translates the graph into kernels, fusing operations where it can, and caches them. Later calls with arguments of the same shapes and dtypes reuse the cached kernels (Figure 13.12).

In PyTorch, `torch.compile` wraps a function or a module. At the first call, its front end, TorchDynamo, executes the function's Python bytecode with *fake tensors* as placeholders and records the tensor operations in a graph (Ansel et al. 2024). AOTAutograd derives the backward graph from the forward graph, and the Inductor compiler turns both into fused kernels, written in Triton for GPUs and in C++ for CPUs. Dynamo also installs *guards*, cheap checks of the assumptions the graph depends on, such as the shapes and dtypes of the inputs. Every later call evaluates the guards and runs the compiled code if they pass.

13.3.1.3 Branches on Values

Tracing cannot follow a branch that depends on a value. A placeholder has a shape but no value, so when the function asks `if x.sum() > 0:`, tracing has no answer, and the answer may differ from call to call. A graph cannot hold Python control flow, which leaves two options. The framework can return the decision to Python at run time, between two graphs.

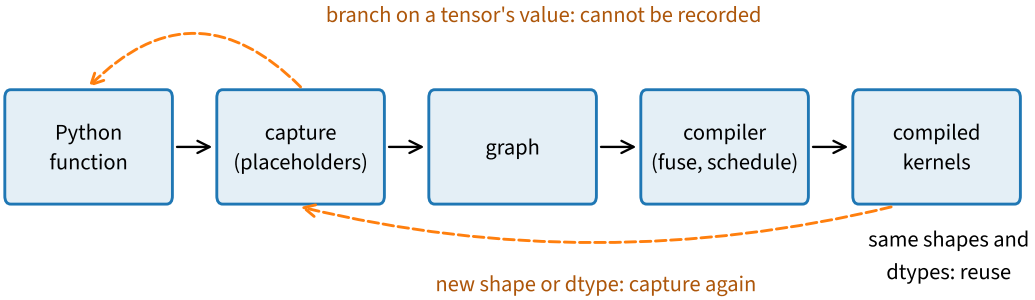


Figure 13.12: Graph capture. The first call traces the function on placeholders; a compiler turns the recorded graph into kernels, which later calls with the same shapes and dtypes reuse. A branch on a tensor's value cannot be recorded, and a new shape or dtype starts a new capture.

Or the program can express the branch as a tensor operation that the graph can hold: a *select* computes both alternatives and keeps one, and a *conditional* operation computes only the one it needs. A branch on a shape, such as `if x.shape[0] > 4:`, causes no difficulty, since shapes are known during tracing. We capture a function that branches on the sign of a sum:

```
def f(x):
    x = x * 2
    if x.sum() > 0:        # Python needs the value of this comparison
        return x + 1
    return x - 1

explanation = torch._dynamo.explain(f)(torch.randn(8, device=d2l.try_gpu()))
print(f'graphs: {explanation.graph_count}, '
      f'breaks: {explanation.graph_break_count}')
for graph in explanation.graphs:
    print(graph.code.strip())
```

```
graphs: 2, breaks: 1
def forward(self, L_x_ : torch.Tensor):
    l_x_ = L_x_
    x = l_x_ * 2;  l_x_ = None
    sum_1 = x.sum()
    gt = sum_1 > 0;  sum_1 = None
    return (gt, x)
def forward(self, L_x_ : torch.Tensor):
    l_x_ = L_x_
    add = l_x_ + 1;  l_x_ = None
    return (add,)
```

Dynamo returns the decision to Python with a *graph break*. The output shows two graphs. The first ends at the comparison and returns its result together with the doubled `x`. Python reads the comparison and takes the branch, and the second graph holds the branch that this random input took, `x + 1` or `x - 1`. A graph break returns control to Python on every call and splits the program into pieces that the compiler optimizes separately. By default Dynamo

does not stop at code it cannot capture: it breaks the graph, runs that code in Python, and prints no warning. `torch.compile(..., fullgraph=True)` turns every break into an error, and `torch._dynamo.explain`, an internal function whose interface may change between releases, counts them. Writing the branch as a select, `torch.where(x.sum() > 0, x + 1, x - 1)`, removes this break.

13.3.1.4 Shapes and Recompilation

A compiler specializes the kernels to the shapes in the graph: it chooses tile sizes, loop bounds, and memory layouts for them. A call with tensors of a new shape or dtype therefore needs a new compilation. A training loop whose last minibatch is smaller than the others, as with the Fashion-MNIST loaders of Section 3.2, compiles once more for that minibatch. A loop over sequences of varying length may compile at almost every step. There are two remedies: keep the set of shapes small, for instance by padding sequences to a few fixed lengths, or compile code that works for a range of sizes. We call a compiled function on vectors of lengths 8, 8, 16, and 32 and count its compilations:

```
compilations = []
def recording_backend(gm, example_inputs):  # Called once per compilation
    compilations.append(gm)
    return gm.forward                       # Run the captured graph as is

g = torch.compile(lambda x: torch.sin(x) + 1, backend=recording_backend)
for n in (8, 8, 16, 32):
    g(torch.ones(n, device=d2l.try_gpu()))
print(f'compilations: {len(compilations)}')
print(compilations[-1].code.strip())
```

```
compilations: 2
def forward(self, s77 : torch.SymInt, L_x_ : torch.Tensor):
    l_x_ = L_x_
    sin = torch.sin(l_x_);  l_x_ = None
    add = sin + 1;  sin = None
    return (add,)
```

`torch.compile` passes each captured graph to a *backend*, normally Inductor. Ours records the graph and runs it without compiling it further, so the list counts compilations. The first call compiles for length 8 and guards on that length, and the second call passes the guard. At length 16 the guard fails and Dynamo compiles again, this time with the length as a symbol: the printed graph takes a `torch.SymInt` argument for it, and the call at length 32 reuses this graph. Handling dynamic shapes this way takes one extra compilation. `torch.compile(..., dynamic=False)` keeps every graph static, and once a function has been compiled eight times (`torch._dynamo.config.recompile_limit`), Dynamo stops compiling it and runs it eagerly.

13.3.2 Operation Fusion

Merging several operations into one kernel is called *fusion*. Its gain can be predicted before anything runs, by counting the bytes that each version moves through memory.

13.3.2.1 Counting Memory Traffic

Python evaluates `gelu_ish(x)` as eight operations, each of which produces a temporary array: $t_1 = 0.5x$, $t_2 = x^3$, $t_3 = 0.04\,t_2$, $t_4 = x + t_3$, $t_5 = 0.8\,t_4$, $t_6 = \tanh t_5$, $t_7 = 1 + t_6$, and the result $t_1 t_7$. In eager execution each operation is a kernel that reads its array arguments from GPU memory and writes its result back. Six operations read one array and two read two, so the chain reads ten arrays and writes eight. A fused kernel reads x once, keeps every temporary in registers, and writes the result once (Figure 13.13). For arrays of n elements of b bytes each, the ratio of the bytes moved is

$$\frac{\text{bytes moved, eager}}{\text{bytes moved, fused}} = \frac{(10+8)\,nb}{(1+1)\,nb} = 9. \tag{13.5}$$

Both versions perform the same arithmetic, about a dozen FLOPs per element, which is far less than the GPU can do while it moves the element's bytes. Both are therefore limited by bandwidth, and fusion should shorten the chain by the ratio of the bytes moved, a factor of nine.

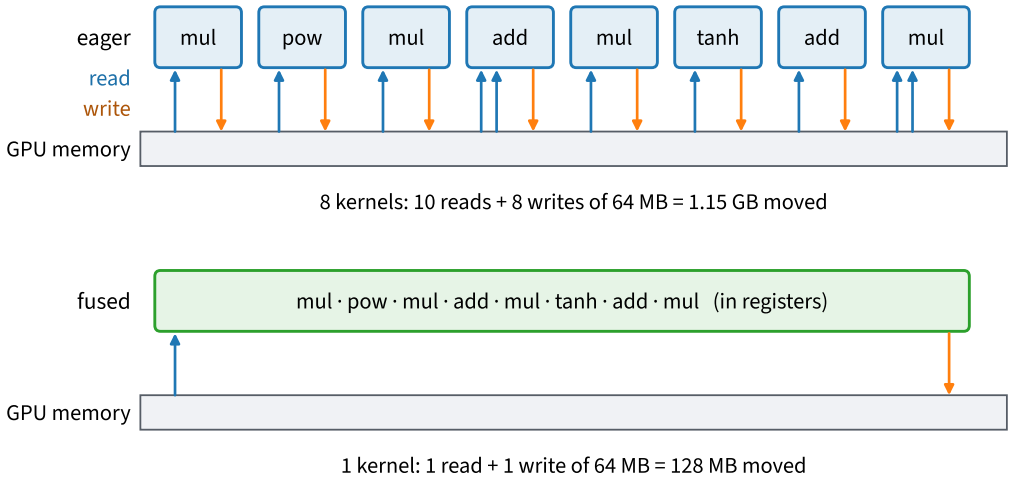


Figure 13.13: Memory traffic of `gelu_ish` on a 64 MB array. Eager execution runs eight kernels that read ten arrays and write eight; the fused kernel reads the input once, keeps the seven intermediate results in registers, and writes the result once.

We compile `d2l.gelu_ish`, the function of Section 13.1, check that the compiled version computes the same values, and time both versions. Each time is then converted into a bandwidth: the bytes that version moves, divided by its time.

```
x = torch.randn(4000, 4000, device=d2l.try_gpu())
compiled = torch.compile(d2l.gelu_ish)
assert torch.allclose(d2l.gelu_ish(x), compiled(x), atol=1e-6)  # Same answer
arrays_moved = {'eager': 18, 'compiled': 2}  # In units of x.nbytes
for desc, fn in (('eager', d2l.gelu_ish), ('compiled', compiled)):
    b = d2l.Benchmark(lambda: fn(x), desc=desc)
    print(f'{b}, {arrays_moved[desc] * x.nbytes / b.time / 1e9:.0f} GB/s')
```

```
eager: 1.25 ms/call, 920 GB/s
compiled: 0.15 ms/call, 878 GB/s
```

The compiled chain is eight to ten times faster, close to the predicted nine, and both versions move data at a similar rate, most of the card's 1 TB/s. The compiler made memory no faster and arithmetic no cheaper. It removed 16 of the 18 array transfers. The cell checks the result before timing it, because a faster kernel is of no use if it computes different values. It uses `allclose`, since the compiler may reorder floating-point operations: the results agree to rounding error, not bit for bit.

The same count applies at the scale of a language model. The chapter's running example is a GPT with a billion parameters and width $d = 2048$, trained on minibatches of 8 sequences of 1,024 tokens (Section 13.1). Give it a GELU MLP of hidden width $4d = 8192$ (`act='gelu'` in `d2l.GPT`). The activation then receives $8192 \cdot 8192 \approx 6.7 \times 10^7$ elements per layer, 134 MB per array in bf16. Written as `gelu_ish`, the eager chain moves 18 such arrays, 2.4 GB, which takes about 2.4 ms at 1 TB/s; the fused chain takes 0.27 ms. The two matrix multiplications around the activation perform $2 \cdot 2 \cdot 8192 \cdot 2048 \cdot 8192 \approx 5.5 \times 10^{11}$ FLOPs, about 3.3 ms at the card's bf16 peak of 165 TFLOP/s. Unfused, the activation adds about three quarters to the time of the MLP; fused, it adds under a tenth. Both times are lower bounds, since real kernels reach neither peak.

The ratio of the two times does not depend on the number of tokens M. The eager chain moves $18{\cdot}4dM$ elements of 2 bytes, and the two matrix multiplications perform $2{\cdot}2{\cdot}M{\cdot}d{\cdot}4d$ FLOPs, so

$$\frac{t_{\text{activation}}}{t_{\text{matmul}}} = \frac{144\,dM/\beta}{16\,d^2M/P} = \frac{9P}{\beta d} \approx 0.72 \tag{13.6}$$

for $P = 165$ TFLOP/s, $\beta = 1$ TB/s, and $d = 2048$. The unfused activation matters less in wider models, since the arithmetic grows with d^2 and the memory traffic only with d.

Real models seldom write GELU as a chain of eight operations. Libraries ship hand-fused kernels for common activations, and `act='gelu'` calls one. A compiler, however, fuses any chain, including ones that no library anticipated: a SwiGLU gate, a custom activation, an optimizer update.

13.3.2.2 Fusion Patterns

Compilers fuse three kinds of patterns. The simplest is the *elementwise chain*, such as `gelu_ish`: a sequence of operations each of which computes element i of its output from element i of its inputs. The whole chain runs in one kernel in which each thread applies every operation to its own elements.

A *reduction*, such as a softmax or a layer normalization, combines elementwise operations with a maximum or a sum over each row. A fused kernel loads a row into on-chip memory once, computes the reduction and the normalized outputs there, and writes the row once.

An *epilogue* is a run of elementwise operations that follows a matrix multiplication, such as adding a bias and applying an activation. The compiler applies them to each tile of the product while the tile is still in registers, before it is written to memory.

Compilers do not change the algorithm. A large matrix multiplication is limited by arithmetic and already runs as a single tuned library kernel; compiling the code around it leaves

it as it is. Nor can fusion remove an intermediate result that the algorithm needs in full. Attention computes a $T \times T$ matrix of scores, normalizes each row with a softmax, and multiplies the result by the values. A compiler fuses the softmax but still writes the score matrix to memory. FlashAttention (Section 10.5) never writes it: it splits the computation into tiles and updates the softmax incrementally as it goes (Dao et al. 2022), a change of algorithm that general-purpose compilers do not find. Such kernels are written by hand, often in Triton (Tillet et al. 2019), a language for GPU kernels embedded in Python.

13.3.3 Compiling a Training Step

A training step contains three kinds of work: matrix multiplications in the forward and backward passes, elementwise chains for the activations and their derivatives, and an elementwise update of every parameter. Compiling the whole step lets the compiler fuse the elementwise work of all three, while the matrix multiplications stay library kernels. Compilation itself takes time, from seconds for a small step to minutes for a large model, and that time is spent at the first call. If compilation takes t_{compile} and the compiled step saves $t_{\text{eager}} - t_{\text{compiled}}$ per call, the compiled run overtakes the eager one after

$$n^* = \frac{t_{\text{compile}}}{t_{\text{eager}} - t_{\text{compiled}}} \tag{13.7}$$

steps. A run much longer than n^* steps gains from compilation, and a shorter one loses. With a compilation time of 30 s and a saving of 10 ms per step, for instance, the break-even point is 3,000 steps, a fraction of a typical training run but more than most quick experiments.

Our step trains a multilayer perceptron with four linear layers of width 1,024 and `gelu_ish` between them. It computes the squared loss of the network's outputs against targets, then the gradients, and updates the parameters by gradient descent.

```
class MLP(nn.Module):
    """Linear layers with the gelu_ish chain between them."""
    def __init__(self, width=1024, depth=4):
        super().__init__()
        self.layers = nn.ModuleList(nn.Linear(width, width)
                                    for _ in range(depth))

    def forward(self, X):
        for layer in self.layers[:-1]:
            X = d2l.gelu_ish(layer(X))
        return self.layers[-1](X)

def train_step(net, X, Y, lr=0.1):  # Forward, backward, SGD update
    loss = ((net(X) - Y) ** 2).mean()
    grads = torch.autograd.grad(loss, list(net.parameters()))
    with torch.no_grad():
        for p, g in zip(net.parameters(), grads):
            p -= lr * g
    return loss
```

`torch.autograd.grad` returns the gradients instead of accumulating them in each parameter's `.grad`, so the step reads the parameters and the data and changes nothing but the parameters. The usual sequence of `zero_grad`, `backward`, and an optimizer's `step` compiles as well.

We compile the step and time its first call, which includes the compilation. Then we time the eager and the compiled step and compute n^* from Equation 13.7:

```
net = MLP().to(d2l.try_gpu())
X, Y = (torch.randn(512, 1024, device=d2l.try_gpu()) for _ in range(2))
compiled_step = torch.compile(train_step)
t_first = d2l.Benchmark(lambda: compiled_step(net, X, Y),
                        warmup=0, repeats=1).time  # Includes compilation
t_eager = d2l.Benchmark(lambda: train_step(net, X, Y)).time
t_compiled = d2l.Benchmark(lambda: compiled_step(net, X, Y)).time
print(f'first call: {t_first:.1f} s\n'
      f'eager: {1000 * t_eager:.2f} ms, compiled: {1000 * t_compiled:.2f} ms\n'
      f'break-even: {t_first / (t_eager - t_compiled):.0f} steps')
```

```
first call: 0.8 s
eager: 2.74 ms, compiled: 1.59 ms
break-even: 713 steps
```

The first call takes far longer than a step, up to several seconds, and the printed n^* is a small fraction of the steps of a real training run.

`torch.compile(train_step)` captures the step in two graphs. Dynamo does not trace into the gradient computation, so `torch.autograd.grad` ends the first graph, which holds the forward pass and the loss. AOTAutograd has already derived the backward graph from it, and Inductor compiled both. After the break, a second graph captures the update. Inductor caches the kernels it compiles on disk, so a second run of this notebook compiles faster than the first.

13.3.4 Kernel Replay

Fusion reduces memory traffic, but it cannot merge matrix multiplications, and each one still needs a launch. Consider a stack of 60 linear layers of width 256 without biases, each followed by a tanh, applied to a minibatch of 64. Each layer performs a matrix multiplication of $2 \cdot 64 \cdot 256 \cdot 256 \approx 8.4 \times 10^6$ FLOPs, which the 4090 completes in a few microseconds, and a tanh over 16,384 elements, which takes about as long. Eager execution launches 120 kernels per call. Python spends 5–15 µs of host time issuing a bare operation (Section 13.2), longer than the GPU needs to run it. The GPU therefore finishes each kernel before Python has issued the next one, and waits. The program is overhead-bound. Fusing the tanh into the preceding matrix multiplication would halve the number of launches, but the remaining 60 would still keep the GPU waiting.

The remedy is to take Python out of the loop: record the sequence of kernel launches once, and replay the whole sequence with a single command. NVIDIA GPUs support this through *CUDA graphs*. A CUDA graph stores each kernel together with its launch configuration and the memory addresses of its arguments. Replaying it takes one launch from the host, after which the GPU runs the kernels back to back. The recording is rigid: its shapes and memory addresses are fixed. New inputs must therefore be copied into the recorded input buffers before a replay, and the outputs of one replay are overwritten by the next. Nothing inside the sequence may wait for a value read by the CPU.

```
deep = nn.Sequential(*[m for _ in range(60)  # 120 small kernels per call
                       for m in (nn.Linear(256, 256, bias=False), nn.Tanh())])
deep = deep.to(d2l.try_gpu())
x = torch.randn(64, 256, device=d2l.try_gpu())

replayed = torch.compile(deep, mode='reduce-overhead')
with torch.no_grad():  # Inference: every call is a complete step
    replayed(x)        # Compiles and warms up; the next call records
    print(d2l.Benchmark(lambda: deep(x), desc='eager'))
    print(d2l.Benchmark(lambda: replayed(x), desc='reduce-overhead'))
```

```
eager: 2.68 ms/call
reduce-overhead: 0.30 ms/call
```

Without Python in the loop, the stack runs many times faster, at roughly the time the GPU needs for its 120 small kernels. Dividing each time by 120 reads off the two costs per kernel: the eager time gives the host time of one launch, and the compiled time gives the GPU time of one kernel, a few microseconds.

`mode='reduce-overhead'` makes `torch.compile` record the compiled kernels as CUDA graphs and replay them. The first call compiles and runs a warm-up, the second records the CUDA graph, and later calls with the same shapes copy the input into the recorded buffer and replay. `Benchmark` warms up with three calls, so the timed calls are replays, seven to eight times faster than the eager calls. The host time per launch of the eager version exceeds that of a bare operation, since every layer is also a call of an `nn.Module`.

We time inference under `torch.no_grad()`, where every call is a complete step and can be replayed as a whole. In training, the forward and the backward pass are recorded as separate graphs and replayed as a pair. Since a replay overwrites the outputs of the previous one, a result that must survive the next call has to be cloned.

13.3.5 Limits of Compilation

Compilation speeds up code that runs many times with the same shapes and is limited by bandwidth or overhead. Outside these conditions it can take more time than it saves.

- **Short runs.** A function called fewer than the n^* times of Equation 13.7 loses time. Quick experiments, evaluation over small datasets, and functions called once are usually faster eager.
- **Branches on values.** A hot loop full of branches on tensor values is captured, if at all, in fragments separated by returns to Python. It gains from compilation only after its branches are rewritten as selects or conditional operations.
- **Changing shapes.** Each new shape compiles again. Pad to a few fixed sizes, compile for symbolic sizes, or leave the function eager.
- **Compute-bound code.** If a profile shows one large matrix multiplication taking most of the time, there is nothing to fuse and no launch overhead to remove. Compiling rarely hurts such code, but it cannot speed it up much.
- **Debugging.** Compiled code does not run line by line. Breakpoints and print statements inside it behave differently, and errors point into generated code. Debug eagerly, then compile.

Compilation removes memory traffic and launches from a program that fits on one GPU. It leaves the largest consumers of memory where they are: the parameters, the optimizer

state, and the activations saved for the backward pass. Section 13.4 measures them and shows how lower precision and recomputation reduce them.

13.3.6 Summary

Eager execution runs a program one operation at a time, so every elementwise operation makes a round trip through GPU memory and every operation needs a launch from Python. Graph capture traces the program on placeholders and records its tensor operations as a graph specialized to the shapes it saw; a branch on a tensor value cannot be recorded, and a new shape requires a new compilation. A compiler fuses elementwise chains, reductions, and the epilogues of matrix multiplications into single kernels, which for `gelu_ish` divides the memory traffic, and with it the time, by about nine. Recording a sequence of kernels as a CUDA graph and replaying it removes the launch overhead of stacks of small kernels. Compilation takes time at the first call, so it saves time only for code that runs often, with fixed shapes, in the bandwidth or overhead regime; it cannot speed up a compute-bound matrix multiplication.

13.3.7 Exercises

1. code **Value-dependent branches.** The function `f` of Section 13.3.1 breaks the graph at its `if`.
    1. Compile `f` with `fullgraph=True` and read the error. Rewrite the branch with `torch.cond`, check with `torch._dynamo.explain` that it captures one graph, and check that both versions return the same values. How does `torch.cond` differ from `torch.where`?
    2. Time the compiled original and the compiled `torch.where` version with `d2l.Benchmark`, once on the 8-element input of Section 13.3.1 and once on a 4000×4000 input. Explain the time difference at each size.
    3. `torch.where` evaluates both branches. Describe a pair of branches for which this costs more than the graph break saves.
2. code **Shape changes.** `torch.compile` guards on the shapes of its inputs and recompiles when a guard fails (Section 13.3.1).
    1. Compile a function that computes `torch.tanh(x) * 2 + 1` with the recording backend of Section 13.3.1, and call it on vectors of lengths 8, 16, 32, and so on up to 8192. Predict the number of compilations, count them, and read the reason for each recompilation with `torch._logging.set_logs(recompiles=True)`.
    2. Repeat with `torch.compile(..., dynamic=False)`. How many compilations happen now, and what do the calls after the last compilation run?
    3. Instead, call `torch._dynamo.mark_dynamic(x, 0)` on the first input before the first call. How many compilations happen, and when would you prefer this over the default?
3. **Padding buckets.** A compiled step that is specialized to static shapes (Section 13.3.1) compiles once for every distinct input shape it sees. Consider a training run of 10^5 steps in which every batch holds sequences of one length ℓ, drawn uniformly from $\{1, \dots, 512\}$, and is padded to a length $\ell' \geq \ell$. One compilation costs 20 s, and a step at padded length ℓ' costs $50\,\text{ms} \cdot \ell'/512$.
    1. Pad every batch to the next multiple of w, for w dividing 512. Find the number

of compilations and the expected padded length as functions of w, and write the total run time $T(w)$.
2. Minimize $T(w)$ over w treated as a continuous variable, and state how the optimal w scales with the compilation cost and with the number of steps. Compare T at $w = 1$, at the two powers of two nearest the optimum, and at $w = 512$.
3. Padding to the next power of two compiles ten shapes. Compute its expected padded length and total run time, and compare them with those of $w = 64$. For which distribution of lengths would powers of two be the better choice?

4. code **Fusion behind a matmul.** ▣ Compilation leaves a large matrix product unchanged but fuses the elementwise chain that follows it. Consider `gelu_ish(a @ a)` for an $n \times n$ matrix `a` in bf16 on a GPU with peak compute P, bandwidth β, and ridge point $I^* = P/\beta$, the arithmetic intensity at which the time to compute equals the time to move the data. Model the product as compute-bound, taking $2n^3/P$ seconds. Model the chain as bandwidth-bound: eager execution moves m arrays of n^2 elements of b bytes each through memory, and the fused kernel moves two such arrays.
   1. Derive the speedup bound $S(n)$ of the compiled over the eager computation in terms of n, m, b, and I^*, and show that it tends to 1 as n grows. With $m = 18$ as counted for `gelu_ish`, evaluate $S(n)$ for bf16 on an RTX 4090 ($I^* \approx 165$ FLOP/byte) at $n = 1024$, 4096, and 16384, and find the n at which $S(n)$ falls to 1.1.
   2. Measure the eager and the compiled time of `gelu_ish(a @ a)` with `d2l.Benchmark` for $n = 256, 512, \dots, 8192$. Plot the measured speedup together with $S(n)$, and explain where and why the two differ.

5. **Fusing a reduction.** ▣ A numerically safe softmax over each row of an $R \times d$ fp32 matrix can be written as five eager operations: the row maximum, the subtraction of that maximum, the exponential, the row sum, and the division by that sum.
   1. Count the elements that the five eager kernels read from and write to memory, keeping only terms proportional to Rd. Count the same for a single fused kernel that holds each row in on-chip memory, and state the speedup bound for a bandwidth-bound softmax.
   2. A row longer than the on-chip memory must be read from memory more than once. Find the smallest number of passes over a row, and the resulting traffic, for a fused kernel that computes the maximum and the sum in separate passes. Then do the same for a kernel that maintains both in one pass over the entries $x_1, \dots, x_d$ of a row, with $m_0 = -\infty$, $s_0 = 0$, and $m_j = \max(m_{j-1}, x_j)$, $s_j = s_{j-1}\, e^{m_{j-1} - m_j} + e^{x_j - m_j}$. Check that $s_d = \sum_j e^{x_j - m_d}$.

6. code **Compiling the optimizer.** ▣ The step of Section 13.3.3 writes plain gradient descent by hand.
   1. Rewrite the step with `torch.optim.Adam` and the usual sequence of `zero_grad`, `backward`, and `step`. Apply `torch._dynamo.explain` to it. How many graphs does it capture, where does it break, and which parts of the step fall into which graph?
   2. Count the parameter-sized arrays that one update must read and write at minimum, for plain SGD and for Adam (Section 9.5). Then time the eager and the compiled step, once with `torch.optim.SGD` and once with `torch.optim.Adam`. For which optimizer does compiling the update gain more, and why?

   Adapted from Michael Lazos, Compiling the optimizer with torch.compile, PyTorch recipe.

7. code **Overhead crossover.** ▣ Use the stack of 60 layers of width 256 at batch size 64

from Section 13.3.4, with its eager and its compiled version. Compilation removes the host time of each launch, about 5–15 µs, from the critical path, but not the time each kernel takes to execute.

1. Time both versions for depths L from 10 to 200 at width 256, and fit each time as $t(L) = c + sL$. Compare the eager slope per kernel with the launch cost, and explain the compiled slope.
2. A kernel that performs F FLOPs and moves B bytes takes at least $\max(F/P, B/\beta)$ seconds on a GPU with peak compute P and bandwidth β. Predict from this bound the width at which the matrix product of one layer takes as long to execute as to launch. Then time both versions at depth 60 for widths from 64 to 4096, plot the ratio of eager to compiled time against the width, and compare the width at which the ratio approaches 1 with your prediction.

8. [extended] **A tracer of your own.** ▯▯ Build a miniature tracing compiler in plain Python, with NumPy arrays as the values on which the captured program finally runs.

   1. Write a class `Tracer` whose instances carry a `shape` but no values, and functions `tanh`, `exp`, and `relu` that take a `Tracer`. Each operation on a `Tracer` appends a node to a shared graph and returns a new `Tracer`. A node holds the operation, its inputs, and the shape of its result. Support the arithmetic operators, the matrix product `@`, the comparison `>`, the methods `sum` and `max` with `axis` and `keepdims` arguments, and the three functions. Trace a copy of `gelu_ish` that calls your `tanh` on a placeholder of a given shape, and print the graph one node per line.
   2. Make `Tracer.__bool__` raise an error. Trace a function that returns `x + 1` when `x.shape[0] > 4` and `x - 1` otherwise, and a second function that branches the same way on `x.sum() > 0`. Explain which of the two your tracer can record, and what the recorded graph keeps of the branch.
   3. Write an interpreter that executes a graph node by node on NumPy arrays, and check its result against a direct NumPy evaluation of `gelu_ish`.
   4. Write a pass that partitions the graph into fused kernels under the rule that an elementwise node may join the kernel of its inputs, a matrix product forms a kernel of its own, and a reduction ends the kernel it belongs to. State how your pass treats a node whose inputs come from different kernels. For `gelu_ish`, for the five-operation softmax, and for the two-layer network `relu(x @ W1 + b1) @ W2 + b2`, count the kernel launches and the array elements moved through memory under one kernel per node and under your partition.

13.4 Memory and Precision

The previous sections asked how fast a training step runs. A step that does not fit in the GPU's memory does not run at all. The models of earlier chapters fit on one card with room to spare, so we never had to ask how much memory a step needs. Large language models change that. Where does the memory of a training step go, and how can a model that barely fits on a card still train on large minibatches?

Consider a GPT with a billion parameters, trained with Adam in fp32 on an RTX 4090, which has 24 GiB of memory. The optimizer keeps four numbers for every parameter: the weight, its gradient, and Adam's two moment estimates. At four bytes each that is 15 GiB before the model has read a single token. The remaining 9 GiB must hold everything that the forward pass stores for the backward pass, and for this model that is several megabytes per token of

the minibatch. Only one to three sequences of 1,024 tokens fit, while language models are trained on minibatches of half a million tokens or more (Brown et al. 2020). A training run that does not fit does not start, and one that barely fits learns from too few examples per step.

The memory of a step splits into two parts: one proportional to the number of parameters, fixed by the model and the optimizer, and one proportional to the number of tokens in the step, the *activations*. We count both for the billion-parameter GPT and check the count against a measurement of the peak. Then we study three techniques that act on the second part. Mixed precision stores activations in 16 bits and runs faster on tensor cores. Activation checkpointing recomputes activations instead of storing them, which takes extra time. Gradient accumulation splits a large minibatch into pieces that fit one at a time. None of the three reduces the memory per parameter; the sections on multiple GPUs show how to divide it among devices.

```
from torch.utils.checkpoint import checkpoint
```

13.4.1 Memory of a Training Step

A training step holds two kinds of tensor. The weights, their gradients, and the optimizer state exist once per parameter, whatever the minibatch. The activations exist once per token of the minibatch. We count each kind for the running example of this chapter, a GPT with a billion parameters, and then ask how many tokens fit next to it on one card.

13.4.1.1 Memory per Parameter

Every parameter of a model that trains with Adam in fp32 carries a weight, a gradient, and Adam's two moment estimates, four bytes each: 16 bytes per parameter, as Section 5.7.3.2 measured. The count depends on the optimizer. Plain SGD keeps a weight and a gradient, 8 bytes; momentum adds a velocity, 12 bytes. The count does not depend on the data: a model with N parameters holds $16N$ bytes whether it processes one token or a million. Mixed precision, below, keeps all four numbers in fp32 and adds a 16-bit copy of every weight for the duration of a step, $18N$ bytes in all. Some systems instead store the weights and gradients in 16 bits and keep an fp32 copy of the weights next to the moment estimates. ZeRO's analysis counts that arrangement as $2 + 2 + 12 = 16$ bytes per parameter (Rajbhandari et al. 2020): the same total, divided differently.

The running example is `d2l.GPT`, the decoder-only transformer of Section 11.2, which stacks pre-norm blocks of causal attention and a SwiGLU feed-forward network between a token embedding and a tied output layer. We give it width $d = 2048$, 16 attention heads, 18 blocks, and the vocabulary of $V = 50{,}257$ tokens and the learned positions of GPT-2. A block holds about $12d^2 = 50$ million parameters, so the blocks hold 0.9 billion and the embedding table another 0.1 billion. Building the model without its data, as in Section 5.2.2.2, confirms the count without allocating 4 GB of weights:

```
config = dict(vocab_size=50257, num_hiddens=2048, num_heads=16,
              num_blks=18, pos='learned')
with torch.device('meta'):  # Shapes and dtypes only, no data
    model = d2l.GPT(**config)
N = d2l.num_params(model)
```

```
print(f'{N / 1e9:.2f} billion parameters, 16N = {16 * N / 2**30:.1f} GiB')
```

```
1.01 billion parameters, 16N = 15.1 GiB
```

13.4.1.2 Memory per Token

Backpropagation needs intermediate results of the forward pass to compute the gradients, so every layer stores some of its inputs or outputs until the backward pass has used them. These stored tensors are the activations. Figure 13.14 follows them through one step. They accumulate during the forward pass and are released one layer at a time during the backward pass, while the gradients appear. Memory therefore peaks at the start of the backward pass, when every activation is still stored.

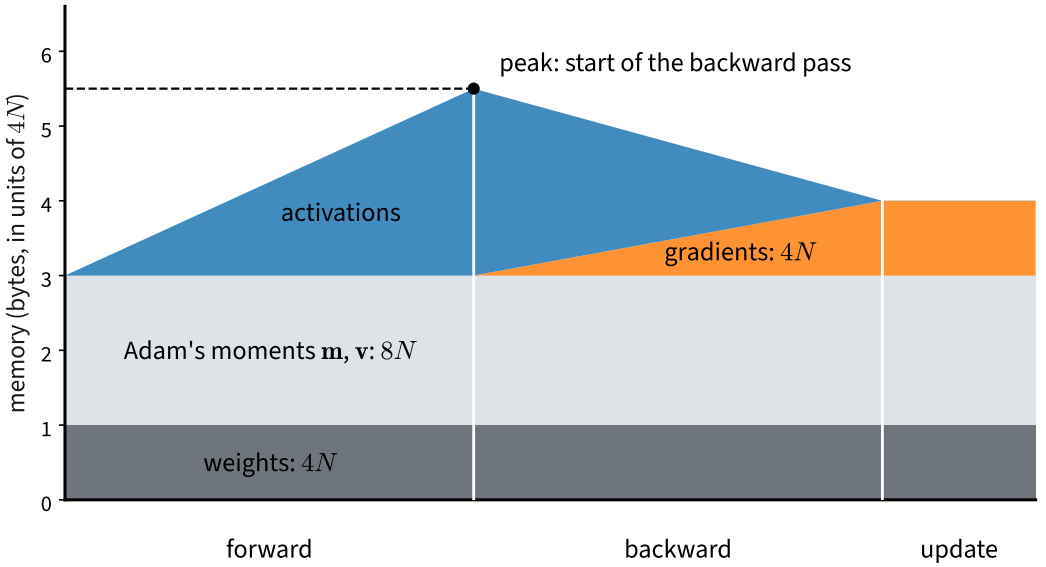


Figure 13.14: The memory of a training step with Adam, in units of $4N$ bytes. The weights ($4N$) and Adam's moment estimates ($8N$) stay for the whole step. Activations accumulate during the forward pass and are released during the backward pass, while the gradients ($4N$) appear. Memory peaks at the start of the backward pass.

How much does one block store per token? We count in units of the width d. The normalization before the attention keeps its input and its output, $2d$ values. The attention keeps the queries, keys, and values, $3d$ values, and its output, another d. The projection after the attention keeps its own copy of that output, d more, and the normalization before the feed-forward network keeps $2d$. The SwiGLU network computes a gate and an up-projection, each of width $\frac{8}{3}d$, and multiplies the activated gate by the up-projection. Its backward pass needs the gate, the up-projection, the activated gate, and the product: four vectors of width $\frac{8}{3}d$, about $11d$. In all, a block keeps about $20d$ values per token, 160 KiB in fp32 at $d = 2048$. Korthikanti et al. (2022) count the same way for the block of GPT-3 without dropout and arrive at $16d$ values besides the attention weights; our feed-forward network keeps more because it has a gate.

Nothing in this count grows with the length T of the sequence, provided that the attention

runs in a fused kernel. The fused attention kernel of Section 10.5 keeps a single number per head and token, the logarithm of the softmax normalizer, and recomputes the attention weights during the backward pass instead of storing them. An attention that stores its $T \times T$ weights keeps at least another $n_h T$ values per token for n_h heads: 16,384 at $T = 1024$, or $8d$, 40% of the rest of the block. An unfused implementation usually stores several such arrays.

The output layer adds a term of its own. The loss needs the log-probability of every token of the vocabulary at every position, so it keeps V values per token, in fp32 for numerical safety (Section 5.5): 196 KiB, more than a whole block.

We check this tally by counting the bytes that the backward pass keeps. The function `saved_bytes` runs the forward pass of a function and adds up the sizes of the arrays it stores for the backward pass. The function `gpt_loss` is the cross-entropy of a GPT's predictions; it uses the input tokens as targets, which avoids loading a dataset and leaves the memory unchanged.

`torch.autograd.graph.saved_tensors_hooks` takes two functions: the first is called on every tensor that autograd saves for the backward pass, the second when the backward pass retrieves it. Ours records the size of the memory block, the *storage*, behind each saved tensor and passes on a detached alias that shares the storage. Passing on the tensor itself would create a reference cycle between a saved output and the graph that produced it, and the graph, with all its tensors, would never be freed. Several saved tensors can be views of one storage, so we count each storage once.

```
def saved_bytes(f, *args):
    """Bytes of the tensors that autograd keeps for the backward pass of f."""
    storages = {}
    def pack(t):  # Called on every tensor saved for the backward pass
        storages[t.untyped_storage().data_ptr()] = (
            t.untyped_storage().nbytes())
        return t.detach()  # Same storage, no reference back to the graph
    with torch.autograd.graph.saved_tensors_hooks(pack, lambda t: t):
        f(*args)
    return sum(storages.values())

def gpt_loss(model, X, bf16=False):
    """Cross-entropy of a GPT's predictions, with the inputs as targets."""
    with torch.autocast(X.device.type, torch.bfloat16, enabled=bf16):
        return F.cross_entropy(model(X).flatten(0, 1), X.flatten())
```

Two subtractions isolate the terms. The count for a minibatch of two sequences minus the count for one sequence gives the bytes of one sequence: the weights, which the backward pass also needs, cancel. The bytes per sequence of a GPT with two blocks minus those of a GPT with one block give the bytes of one block. What remains of the one-block model is the embedding, the output layer, and the loss.

```
def bytes_per_token(config, bf16=False, num_steps=1024):
    """Bytes stored per token by one block, and by the rest of a GPT."""
    gpu, per_token = d2l.try_gpu(), []
    for L in (1, 2):  # GPTs with one and with two blocks
        model = d2l.GPT(**{**config, 'num_blks': L}).to(gpu)
        X1, X2 = (torch.randint(0, config['vocab_size'], (b, num_steps),
```

```
                                device=gpu) for b in (1, 2))
            per_token.append((saved_bytes(gpt_loss, model, X2, bf16)
                              - saved_bytes(gpt_loss, model, X1, bf16))
                             / num_steps)
        return per_token[1] - per_token[0], 2 * per_token[0] - per_token[1]

    block, head = bytes_per_token(config)
    print(f'one block: {block / 2**10:.0f} KiB per token '
          f'({block / (4 * config["num_hiddens"]):.1f} fp32 values of width d)')
    print(f'embedding, output layer, and loss: {head / 2**10:.0f} KiB per token')
```

```
one block: 149 KiB per token (18.7 fp32 values of width d)
embedding, output layer, and loss: 212 KiB per token
```

A block keeps the tally plus whatever intermediate results its kernels keep in addition. A normalization or an activation function written as several operations keeps a few vectors that a fused kernel would not. The embedding, the output layer, and the loss keep a little more than the 196 KiB of log-probabilities.

A block keeps about 19 values of width d per token, close to the tally: `scaled_dot_product_attention` runs the fused kernel, which stores only the logarithm of the normalizer, and the normalizations keep little beyond their inputs and outputs. The embedding, the output layer, and the loss keep about 210 KiB per token.

13.4.1.3 The Memory Budget

The two counts give the memory of a training step with n tokens. With a bytes per token for each of the L blocks and h bytes per token for the rest of the model,

$$M(n) \approx 16N + n\,(La + h), \tag{13.8}$$

and a step fits on a GPU with M_{GPU} bytes if $M(n) \le M_{\text{GPU}}$. The first term is fixed by the model and the optimizer, the second grows with the minibatch. The largest minibatch is therefore $n_{\max} = (M_{\text{GPU}} - 16N)/(La + h)$.

The equation counts the gradients, $4N$ bytes, as present for the whole step. That is exact under gradient accumulation (Section 13.4.4), where the gradients of one micro-batch stay in memory while the next micro-batch runs its forward pass. A single step that frees its gradients after the update peaks lower, at $12N$ bytes plus the activations, because its gradients appear only while the activations are released (Figure 13.14). Exercise 1 locates the peak in both cases. The budgets of this section are those of a step with accumulation.

`zero_grad(set_to_none=True)` releases the gradients after each update, so a step without accumulation reaches $12N$ plus the activations. With accumulation, the gradients are added up in `.grad` and live through every micro-batch.

For our GPT, $16N = 15.1$ GiB and the card has 24 GiB, which leaves 8.9 GiB for the activations.

With $a \approx 0.15$ MiB and $h \approx 0.21$ MiB from the count, each token needs $La + h \approx 2.8$ MiB, which leaves room for about 3,200 tokens, a little over three sequences of 1,024 tokens.

These estimates are optimistic: the allocator rounds and fragments its blocks, and libraries reserve workspace of their own.

13.4.1.4 Measured Peak Memory

A count is a model of what the framework stores, and a model can leave things out. We check Equation 13.8 on a GPT that is small enough to measure on a card with 8 GiB: the width, the vocabulary, and the block of the running example, but only two blocks. We measure the peak memory of the forward and backward pass with 1,024, 2,048, and 4,096 tokens per step, in sequences of 1,024 tokens as in the count, and compare the growth per token with the count of the previous cell. With few tokens, other parts of the step can set the peak, so we take the growth between 2,048 and 4,096 tokens.

PyTorch's caching allocator counts the bytes in live tensors, and `max_memory_allocated` reports the highest count since the last `reset_peak_memory_stats`. `d2l.peak_memory` (Section 5.7) reads the peak of a call relative to the memory allocated before it, which here holds the weights.

```
gpu, V = d2l.try_gpu(), config['vocab_size']
model = d2l.GPT(**{**config, 'num_blks': 2}).to(gpu)
gpt_loss(model, torch.randint(0, V, (1, 1024), device=gpu)).backward()
peaks = {}  # The step above was a warm-up
for n in (1024, 2048, 4096):  # Tokens per step, in sequences of 1,024
    X = torch.randint(0, V, (n // 1024, 1024), device=gpu)
    model.zero_grad(set_to_none=True)
    peaks[n] = d2l.peak_memory(lambda: gpt_loss(model, X).backward())
    print(f'{n:5d} tokens: peak {peaks[n] / 2**30:.2f} GiB above the weights')
slope = (peaks[4096] - peaks[2048]) / (4096 - 2048)
print(f'measured: {slope / 2**10:.0f} KiB per token; '
      f'counted: {(2 * block + head) / 2**10:.0f} KiB per token')
```

```
 1024 tokens: peak 1.53 GiB above the weights
 2048 tokens: peak 1.77 GiB above the weights
 4096 tokens: peak 3.53 GiB above the weights
measured: 905 KiB per token; counted: 511 KiB per token
```

The peak grows with the number of tokens, as Equation 13.8 predicts, but its slope is not the count of stored bytes. The loss adds a term of its own. At the start of the backward pass, it creates a gradient for each of the V logits of every token while the log-probabilities are still stored. It passes through up to two such fp32 arrays, 196 KiB per token each, before the log-probabilities are released. At the peak, a token therefore needs up to $8V$ bytes more than the count.

The measured slope exceeds the count by 0.38 MiB per token, two fp32 arrays of the size of the log-probabilities. At the peak the billion-parameter model thus needs about 3.2 MiB per token, which leaves room for about 2,800 tokens. With 1,024 tokens the peak comes later in the step, when the gradients of the weights exist, and it lies above the line through the other two points.

The allocator's own record shows the course of Figure 13.14 in a real step. `torch.cuda.memory._record_memory_history()` records every allocation and release from then on, and `torch.cuda.memory._snapshot()` returns the record. We record one complete training step with Adam on 4,096 tokens, after a first step has created the moment estimates, and note where the forward and the backward pass end:

```
X = torch.randint(0, V, (4, 1024), device=gpu)
opt = torch.optim.Adam(model.parameters())
gpt_loss(model, X).backward()  # A first step creates the moment estimates
opt.step()
opt.zero_grad(set_to_none=True)

events = lambda: torch.cuda.memory._snapshot()['device_traces'][0]
start = torch.cuda.memory_allocated()
torch.cuda.memory._record_memory_history()
loss = gpt_loss(model, X)
forward_end = len(events())
loss.backward()
backward_end = len(events())
opt.step()
opt.zero_grad(set_to_none=True)
trace = events()
torch.cuda.memory._record_memory_history(enabled=None)
```

Adding up the sizes of the allocations and subtracting those of the releases, starting from the memory allocated before the step, gives the allocated memory after every event:

```
sign = {'alloc': 1, 'free_completed': -1}
sizes = torch.tensor([sign.get(e['action'], 0) * e['size'] for e in trace])
series = ((start + sizes.cumsum(0)) / 2**30).tolist()
bounds = [0, forward_end, backward_end, len(trace)]
phases = list(zip(bounds[:-1], bounds[1:]))
d2l.plot([list(range(a, b)) for a, b in phases],
         [series[a:b] for a, b in phases], 'allocation event',
         'allocated (GiB)', legend=['forward', 'backward', 'optimizer step'])
```



The trace starts at the weights and moment estimates, $12N$. Memory climbs through the forward pass as the blocks store their activations and jumps when the output layer computes the logits and their log-probabilities. The peak comes at the start of the backward pass, with the gradient of the logits. Then the activations are released while the gradients appear,

the optimizer step briefly allocates temporaries of the size of the gradients, and `zero_grad` releases the gradients.

13.4.2 Mixed Precision

Mixed-precision training, introduced in Section 5.5, keeps the weights in fp32 and runs the matrix multiplications of the forward and backward pass in a 16-bit format, with the loss and other reductions in fp32. bf16 has the exponent range of fp32, so its gradients do not underflow; fp16 has a narrower range and needs loss scaling (Section 5.5.3.3). Mixed precision changes both terms of Equation 13.8, and it changes the time of a step. We measure the memory first. The count of the previous subsection, repeated with bf16 arithmetic, gives the bytes per token:

```
block16, head16 = bytes_per_token(config, bf16=True)
print(f'one block: {block16 / 2**10:.0f} KiB per token in bf16, '
      f'{block16 / block:.2f} of fp32')
print(f'embedding, output layer, and loss: {head16 / 2**10:.0f} KiB per token')
```

```
one block: 87 KiB per token in bf16, 0.58 of fp32
embedding, output layer, and loss: 306 KiB per token
```

A block stores well over half of its fp32 bytes, not half. The loss still keeps its log-probabilities in fp32, so the term h hardly changes. And the 16-bit copies of the weights, $2N$ bytes, stay in memory until the backward pass has used them. For the billion-parameter GPT the fixed term grows from $16N = 15.1$ GiB to $18N = 17.0$ GiB, and the room for activations shrinks from 8.9 to 7.0 GiB. The smaller activations and the added weight copies largely offset each other, and mixed precision raises the largest minibatch by a quarter at most.

Under `torch.autocast`, the residual stream that connects the blocks stays in fp32, and so do the normalizations, which autocast runs in fp32 for accuracy. Each matrix multiplication keeps its own bf16 copy of its input, next to the fp32 output of the normalization that produced it. A block stores about 0.6 of its fp32 bytes, and the memory per token at the peak falls from about 3.2 MiB to 2.0 to 2.2 MiB, depending on how many fp32 arrays of the size of the logits the loss holds at once. The largest minibatch grows by 15 to 25%, to 3,200 to 3,500 tokens.

Memory is the smaller of the two gains. The larger one is time: the tensor cores of the RTX 4090 multiply bf16 matrices at twice the rate of fp32 ones (Section 13.2), and each operand takes half the bytes. Whether a plain fp32 matrix multiplication uses the tensor cores as tf32, with fp32 range and a 10-bit mantissa, is a framework default (Section 5.5). A speedup quoted against true fp32 thus mixes two effects. To separate them, we time one training step of a three-layer MLP of width 8192, on a minibatch of 2,048 examples, in three configurations: true fp32, tf32, and bf16.

```
mlp = nn.Sequential(nn.Linear(64 * 64, 8192, bias=False), nn.ReLU(),
                    nn.Linear(8192, 8192, bias=False), nn.ReLU(),
                    nn.Linear(8192, 10, bias=False)).to(gpu)
for layer in mlp[::2]:  # Weights drawn from N(0, 0.02^2)
    nn.init.normal_(layer.weight, std=0.02)
X_mlp = torch.randn(2048, 64 * 64, device=gpu)
```

```
y_mlp = torch.randint(0, 10, (2048,), device=gpu)
opt = torch.optim.SGD(mlp.parameters(), lr=0.01)

def step(bf16):
    with torch.autocast(X_mlp.device.type, torch.bfloat16, enabled=bf16):
        l = F.cross_entropy(mlp(X_mlp), y_mlp)
    l.backward()
    opt.step()
    opt.zero_grad(set_to_none=True)

torch.set_float32_matmul_precision('highest')  # True fp32
print(d2l.Benchmark(lambda: step(False), desc='fp32'))
torch.set_float32_matmul_precision('high')  # fp32 matmuls may use tf32
print(d2l.Benchmark(lambda: step(False), desc='tf32'))
print(d2l.Benchmark(lambda: step(True), desc='bf16'))
```

```
fp32: 22.60 ms/call
tf32: 14.60 ms/call
bf16: 9.47 ms/call
```

The step gets faster twice. tf32 runs the fp32 matrix multiplications on the tensor cores and shortens the step by about a third, without changing a single stored byte. bf16 then runs about 1.5 to 2 times as fast as tf32, close to the factor of two in arithmetic rate and bytes. The fair baseline for a speedup from mixed precision is tf32, not true fp32.

For matrix multiplications PyTorch defaults to `'highest'`, true fp32; `torch.set_float32_matmul_precision('high')` allows tf32. Convolutions follow a separate switch, `torch.backends.cudnn.allow_tf32`, which is on by default. The bf16 step needs no `GradScaler`, since bf16 gradients do not underflow; with fp16, it would (Section 5.5.3.3).

13.4.3 Activation Checkpointing

Mixed precision leaves the running example at one to three sequences of 1,024 tokens per step. Activation checkpointing (Chen et al. 2016), introduced in Section 5.7.3.3, goes further. It stores only the inputs of selected segments of the network, and during the backward pass it reruns the forward computation of each segment to regenerate the activations that segment needs. Every checkpointed segment thus runs its forward pass twice. A backward pass takes about twice as long as a forward pass, so a step then takes about as long as four forward passes instead of three.

A transformer comes in natural segments, its blocks. With one segment per block, each block stores only its input, d values or $4d$ bytes per token in fp32, instead of its a bytes of activations. During the backward pass one block at a time is recomputed, so the activations of a single block return while it is processed. The memory per token at the peak falls to at most

$$L \cdot 4d + a + h + 8V,$$

the stored inputs of all blocks, one recomputed block, and the output layer and the loss together with the gradient of the logits. The sum is an upper bound, because the gradient of the logits is released before the first block is recomputed. For the billion-parameter GPT in fp32 the first term is 18×8 KiB $= 144$ KiB, and the sum is about 1 MiB per token.

With $a \approx 150$ KiB and $h \approx 210$ KiB, the sum is 0.9 MiB per token, and the largest minibatch grows to about 10,000 tokens, more than three times as many as without checkpointing.

The output layer and the loss now store more per token than all 18 blocks together. Computing the loss in chunks of tokens, so that the log-probabilities of only one chunk exist at a time, removes most of that term; Exercise 8 builds it.

We measure the effect of checkpointing on a stack of four blocks of the running example, with two sequences of 1,024 tokens. The cell reports the bytes stored per token, the peak memory beyond the weights and gradients, and the time of a forward and backward pass.

```
blocks = d2l.GPT(**{**config, 'num_blks': 4}).to(gpu).blks
def stack(H, remat):
    for blk in blocks:
        H = checkpoint(blk, H, use_reentrant=False) if remat else blk(H)
    return H.square().mean()

d = config['num_hiddens']
H1, H2 = (torch.randn(b, 1024, d, device=gpu, requires_grad=True)
          for b in (1, 2))
for remat in (False, True):
    stored = (saved_bytes(stack, H2, remat)
              - saved_bytes(stack, H1, remat)) / 1024
    stack(H2, remat).backward()  # Warm-up; allocates the gradients
    peak = d2l.peak_memory(lambda: stack(H2, remat).backward())
    t = d2l.Benchmark(lambda: stack(H2, remat).backward()).time
    print(f'checkpointing={remat!s:5}: stores {stored / 2**10:4.0f} KiB per '
          f'token, peak {peak / 2**30:.2f} GiB, {1000 * t:.0f} ms per step')
```

```
checkpointing=False: stores  606 KiB per token, peak 1.25 GiB, 46 ms per step
checkpointing=True : stores   40 KiB per token, peak 0.42 GiB, 57 ms per step
```

Checkpointing reduces what four blocks store per token from four blocks' worth of activations to five vectors of width d, 40 KiB: the inputs of the four blocks and the output that the loss reads. The peak falls by less, because the backward pass recomputes one block at a time and holds its activations while it does. The step takes a fifth to a third longer, within the bound of one extra forward pass. For a fixed minibatch, checkpointing always adds time; it is the right choice only when memory, not speed, limits the minibatch.

`checkpoint(blk, H, use_reentrant=False)` runs `blk` without storing its intermediate results and keeps what it needs to rerun it, including the state of the random number generator, so that dropout draws the same mask twice. PyTorch recommends `use_reentrant=False`, the implementation shown here, and warns when the argument is missing. The peak falls by a factor of about three.

Per-block checkpointing recomputes everything in a block, including its matrix multiplications, which take most of its time. Much of the memory of a block, in contrast, sits in results that are cheap to compute. *Selective recomputation* stores the expensive results and recomputes the cheap ones. Korthikanti et al. (2022) recompute only the core of the attention: the scores $\mathbf{QK}^\top$, their softmax, and its product with the values, which store $T \times T$ values per head but take a small part of the arithmetic. The fused attention kernel does the same inside one kernel: it keeps the queries, keys, and values and recomputes the attention weights. Framework policies generalize the idea: keep the outputs of the matrix multiplications with the weights, and recompute the elementwise operations. Exercise 6 implements

such a policy for our block.

In PyTorch, `torch.utils.checkpoint.create_selective_checkpoint_contexts` turns a policy, a function that decides for each operation whether to keep its output, into the `context_fn` argument of `checkpoint`.

13.4.4 Gradient Accumulation

With checkpointing, the running example fits several thousand tokens per step, still far fewer than the half a million tokens per minibatch it should train with. Gradient accumulation, described in Section 9.3.2, closes the rest of the gap without storing more activations. It splits the minibatch the optimizer should see, the *global* batch, into m micro-batches, runs the forward and backward pass on each while adding up their gradients, and takes one optimizer step at the end. Only one micro-batch's activations are stored at any time, so the peak memory follows the micro-batch, while the update sees the global batch: $B_{\text{global}} = m\, B_{\text{micro}}$. For the running example, micro-batches of four sequences of 1,024 tokens and $m = 128$ give a global batch of half a million tokens.

The accumulated gradient must equal the gradient of the global batch. If the loss averages over the examples of a micro-batch, as the mean squared error and the cross-entropy do, each micro-batch loss is divided by m, so that the sum of the gradients becomes the average over the global batch. We check this on a linear model with four micro-batches of 32 examples:

```
net = nn.Linear(1024, 1, bias=False).to(gpu)
X = torch.randn(128, 1024, device=gpu)
y = torch.randn(128, 1, device=gpu)
loss = nn.MSELoss()

net.zero_grad()  # Global batch: one backward pass over all 128 examples
loss(net(X), y).backward()
full = net.weight.grad.clone()

m = 4  # Micro-batches of 128 / m = 32 examples; gradients add up in .grad
net.zero_grad()
for Xb, yb in zip(X.chunk(m), y.chunk(m)):
    (loss(net(Xb), yb) / m).backward()  # / m because MSELoss averages
print(f'max gradient difference: {(full - net.weight.grad).abs().max():.1e}')
```

```
max gradient difference: 1.2e-07
```

The two gradients agree up to rounding. The equivalence rests on conditions that this example meets without effort. The loss must add up over examples, and the micro-batches must have equal sizes, or their losses must be weighted by their sizes. Layers that couple the examples of a batch, batch normalization above all, see only the micro-batch. And everything that acts on the whole gradient, such as clipping and the optimizer step, must wait for the last micro-batch.

Accumulation leaves the number of arithmetic operations unchanged. It can slow a step only by lowering efficiency. Each micro-batch reads all the weights once in the forward and once in the backward pass, and a small micro-batch does too little arithmetic per byte read to keep the tensor cores busy (Section 13.1). A micro-batch of thousands of tokens is far into the compute-bound regime, where accumulation adds almost no time. Accumulation

never makes a step faster: it changes when the optimizer steps, not how much work each token needs.

13.4.5 Choosing a Memory Technique

The techniques of this section address different terms of Equation 13.8, and each has a drawback. Table 13.4 lists them by the symptom that calls for them, with the term that each changes and its cost.

Table 13.4: Memory techniques by symptom, with their effect on Equation 13.8 and their cost.

symptom	technique	effect on memory	cost
a step is slow and compute-bound	bf16 mixed precision	fewer bytes per activation, though not half; $+2N$ for weight copies	rounding; fp16 also needs loss scaling
one sequence barely fits, or not at all	activation checkpointing	stored activations per block from a to $4d$ bytes per token	one more forward pass, a fifth to a third more time
the logits dominate the activations	a loss computed in chunks of tokens	h for one chunk at a time	one more pass through the output layer
the minibatch is too small for training	gradient accumulation	none; the batch per update grows m-fold	efficiency of the smaller micro-batches
$16N$ alone fills the card	sharding the weights, gradients, and optimizer state over GPUs	$16N/k$ on each of k GPUs	communication

For the running example on one RTX 4090 these steps combine into one recipe: bf16 for speed, checkpointing of every block for micro-batches of four sequences of 1,024 tokens, and accumulation over 128 micro-batches for a global batch of half a million tokens. The model then trains, but every step still holds the 17 GiB of weights, gradients, moment estimates, and bf16 weight copies that none of the three techniques touches. Data parallelism, the subject of the next section, makes a full copy of them on every GPU. Dividing them among the GPUs instead is the subject of Section 13.6.3.

13.4.6 Summary

The memory of a training step has a part that grows with the parameters, $16N$ bytes with Adam in fp32, and a part that grows with the tokens of the minibatch, the activations that the forward pass stores for the backward pass. Counting the saved tensors gives about $20d$ values per token for a transformer block with fused attention, more for an attention that stores its $T \times T$ weights, plus the log-probabilities of the whole vocabulary for the loss. For a GPT with a billion parameters on a 24 GiB card this leaves room for one to three sequences of 1,024 tokens per step, and a measurement of the peak checks the count. Mixed precision shrinks the activations, though not to half, and speeds up the matrix multiplications, but

adds 16-bit weight copies. Activation checkpointing stores only the input of each block and recomputes the rest, which fits several times as many tokens for a fifth to a third more time. Gradient accumulation then builds a large global batch from micro-batches that fit, and it slows the step only when those micro-batches are too small to keep the GPU busy.

13.4.7 Exercises

1. **Where the peak lands.** Figure 13.14 places the peak of a training step at the start of the backward pass. Consider a stack of n identical layers. The parameters of each layer occupy w bytes, and its forward pass stores a bytes of activations. Assume that no gradients exist when the backward pass starts, and that the backward pass of a layer frees the activations of that layer and creates its parameter gradient of w bytes. Leave out the parameters and the optimizer states, whose sizes stay constant.
    1. Write the bytes held by activations and gradients after the backward pass has processed j of the n layers. Show that the peak lies at the start of the backward pass when $a > w$ and at its end when $a < w$.
    2. A block of the running example holds about $12d^2$ parameters and stores about $20d$ fp32 values per token. For how many tokens per step does the peak lie at the start of the backward pass? Is that a realistic number of tokens for training?
    3. Under gradient accumulation over m micro-batches, the gradients of the first micro-batch stay allocated while the later micro-batches run. Find the peak with accumulation, and the values of m for which accumulation lowers the peak below that of a single step on the full batch.
2. code **Optimizer-step memory.** The trace of Section 13.4.1.4 shows that the optimizer step allocates temporaries of its own. Build the two-block GPT of that subsection three times, each with a different `torch.optim.Adam`: with `foreach=False`, with the default arguments, and with `fused=True`.
    1. Run two training steps with each optimizer. On the second step, record `torch.cuda.memory_allocated()` just before `opt.step()`, reset the peak statistics, and record `torch.cuda.max_memory_allocated()` after `opt.step()`. Express the extra bytes of each update in units of the gradient size $4N$.
    2. The three implementations update one parameter tensor at a time, apply each elementwise operation to all parameter tensors at once, or run the whole update in one kernel per group of tensors. Explain the three measurements from this difference. When does the optimizer step, rather than the start of the backward pass, set the peak of a training step?
3. **Clipping under accumulation.** Gradient clipping to norm τ replaces a gradient $\mathbf{g}$ by $\mathbf{g}\min(1, \tau/\|\mathbf{g}\|)$. Let $\mathbf{g}_1, \dots, \mathbf{g}_m$ be the micro-batch gradients after the scaling by $1/m$ of Section 13.4.4, so that their sum $\mathbf{g}$ is the gradient of the global batch.
    1. A training loop clips each $\mathbf{g}_i$ before adding it to the accumulator. Find the largest norm that the accumulated gradient can reach, and a condition under which the loop gives the same result as clipping $\mathbf{g}$ once before the optimizer step.
    2. Under fp16 loss scaling with factor S, the accumulator holds $S\mathbf{g}$ once the last micro-batch has been added. In which order must the loop divide by S, clip, and step? What goes wrong if it clips before dividing, and what goes wrong if S changes before the last micro-batch has been added?

Adapted from the PyTorch documentation, Automatic Mixed Precision examples, sections "Gradient clipping" and "Gradient accumulation".

4. code **fp16 gradient window.** ▣ The 5-bit exponent of fp16 gives it normal numbers from 2^{-14} to 65,504 and subnormal numbers down to 2^{-24}. Loss scaling (Micikevicius et al. 2018) multiplies the loss by a factor S before the backward pass and divides the gradients by S afterwards. Use `mlp`, `X_mlp`, and `y_mlp` as defined in Section 13.4.2, and take as reference the fp32 parameter gradients computed under `torch.set_float32_matmul_precision('highest')`.
    1. Plot a histogram of $\log_2 |g|$ over the nonzero entries g of the reference gradients. Which fractions of the entries lie below 2^{-14} and below 2^{-24}, and which factors S place 99.9% of the scaled entries Sg in the normal range of fp16?
    2. For $S = 2^0, 2^2, \dots, 2^{24}$, compute the gradients of S times the loss under `torch.autocast('cuda', dtype=torch.float16)`, divide them by S, and plot their relative error $\|\hat{\mathbf{g}} - \mathbf{g}\|/\|\mathbf{g}\|$ with respect to the reference $\mathbf{g}$, marking the runs with non-finite entries. Repeat the sweep with `torch.bfloat16`. In which range of S is fp16 more accurate than bf16, and why is that possible?
    3. `torch.amp.GradScaler` starts at $S = 2^{16}$. Where does this factor lie in the window of the previous sub-problem? The gradient of the loss with respect to a logit is the difference between the predicted and the target probability, divided by the batch size, here 2048. How does the window move if the batch grows to 32,768 examples? Explain why a static loss scale that works for one workload can fail for another.

5. **Checkpoint placement.** ▣ Consider a stack of n identical residual blocks, and measure memory in units of the input of one block. The backward pass of one block needs c units of stored tensors, its input included. A schedule with segments of s consecutive blocks stores only the input of each segment during the forward pass. During the backward pass it recomputes one segment at a time, while the stored inputs of the earlier segments remain in memory.
    1. Derive the peak activation memory as a function of s, and the s that minimizes it. Take $c = 20$ for a block of the running example and $n = 18$ blocks. Compare storing everything, the per-block checkpointing of Section 13.4.3, and the best segment length, in memory and in the number of block forward evaluations per step.
    2. Chen et al. (2016) reduce the memory further by recursion. The schedule stores the input of the stack and the input of its second half, processes the second half recursively, and then processes the first half recursively from its stored input. For n a power of two, derive the peak memory and the total number of block forward evaluations, the original forward pass included. For $n = 16$ and $n = 1024$, compare both quantities with those of the best segment length.

6. code **Selective recomputation.** ▣ Per-block checkpointing stores only the input of each block. The function `torch.utils.checkpoint.create_selective_checkpoint_contexts`, passed to `checkpoint` through its `context_fn` argument, instead caches the outputs of chosen operations during the forward pass and recomputes the remaining operations. Its policy function receives each operation together with its arguments. The linear layers of `d2l.GPT` have no biases, and each dispatches to `torch.ops.aten.mm.default`.
    1. For one block of the running example, list the tensors stored between the forward and the backward pass, and the matrix multiplications that the backward pass

recomputes, under three policies: per-block checkpointing, caching the outputs of every `mm`, and caching all outputs except those of the feed-forward network's gate and up-projection.
2. Implement the last two policies for the stack of four blocks of Section 13.4.3, and measure the stored bytes per token, the peak memory, and the step time as that subsection does. Which policy would you choose for the running example, and why?

7. code **Cost of accumulation.** ▯ Split the global batch of 2048 examples of the bf16 configuration of Section 13.4.2 into m equal micro-batches, and accumulate their gradients, for $m = 1, 2, 4, \dots, 64$.
   1. Let the network have N parameters. A micro-batch of B_{micro} examples performs about $6NB_{\text{micro}}$ floating-point operations in its forward and backward passes, and it moves at least $12N$ bytes: it reads the bf16 weights in each of the two passes, and it reads and writes the fp32 gradient accumulator. Ignore the activations. Take from Table 13.2 the peak arithmetic rate and the memory bandwidth of the RTX 4090, predict the time per global batch as a function of m, and find the m beyond which the micro-batches are bandwidth-bound.
   2. Measure the time per global batch for every m, and plot it together with the prediction. Identify the weight and gradient traffic of the bf16 configuration that the count of $12N$ bytes leaves out.
   3. Suppose that activation checkpointing let each micro-batch hold four times as many examples, with a third more arithmetic. Under the model of the first subproblem, for which $m \geq 4$ of the sweep does accumulation over $m/4$ checkpointed micro-batches take less time than accumulation over m plain ones?

8. [extended] **The loss in chunks.** ▯▯ Under per-block checkpointing, the output layer and the loss of the running example store more per token than all 18 blocks together, because the loss keeps $V = 50{,}257$ log-probabilities per token and the backward pass creates their gradient.
   1. Write the output layer and the cross-entropy so that they process the tokens of a step in chunks of n_c tokens, and wrap each chunk in a checkpoint, so that the log-probabilities of only one chunk exist at a time. Check that the loss and the gradients match the unchunked version up to rounding.
   2. Using the counts of this section, predict the largest micro-batch of the running example on a 24 GiB card, in fp32 with per-block checkpointing, with and without the chunked loss, for $n_c = 1024$. Which term of Equation 13.8 limits the micro-batch now?
   3. On the two-block GPT of Section 13.4.1.4, measure the growth of the peak memory per token with the chunked loss, as that subsection does, and compare it with your prediction. How much time does the chunked loss add per step?

13.5 Multi-GPU from First Principles

Every training run so far has used one GPU, although this book's build machine holds four. A run on one of them leaves three quarters of the machine's arithmetic idle. The simplest way to use all four is to give every GPU a copy of the model and a quarter of each minibatch. The copies must then agree after every step, so the GPUs exchange their gradients, and the exchange takes time that a single GPU never spends. When does a second GPU shorten a

step, and when does the exchange cost more than it saves?

On our machine the exchange is slow compared with the work. Each RTX 4090 performs about 83 trillion fp32 operations per second and reads its own memory at 1 TB/s. Gradients travel between the cards over PCIe and through host memory, because the cards cannot address each other directly (Section 13.2.4.2). A PCIe 4.0 link carries a nominal 32 GB/s in each direction, about thirty times less than a GPU reads from its own memory. A second GPU shortens training only if the computation it takes over lasts longer than the exchange it adds.

This section builds the scheme, called *data parallelism*, by hand. We split each minibatch, compute gradients on every device, sum them with a collective operation called an *allreduce*, and apply the same update everywhere. We then derive the allreduce algorithm that collective libraries use. Finally, a cost model predicts the time of a step from three quantities that we measure: the time a device needs for its share of the minibatch, the fixed cost of one allreduce, and the bandwidth between devices. Our test case is LeNet on Fashion-MNIST. Its steps are so short that fixed costs dominate them, which makes it the hardest case for data parallelism and the easiest one to take apart.

```
import time
```

```
torch.set_float32_matmul_precision('high')
```

13.5.1 Three Ways to Split

A model and its minibatch can be divided among devices in three ways (Figure 13.15).

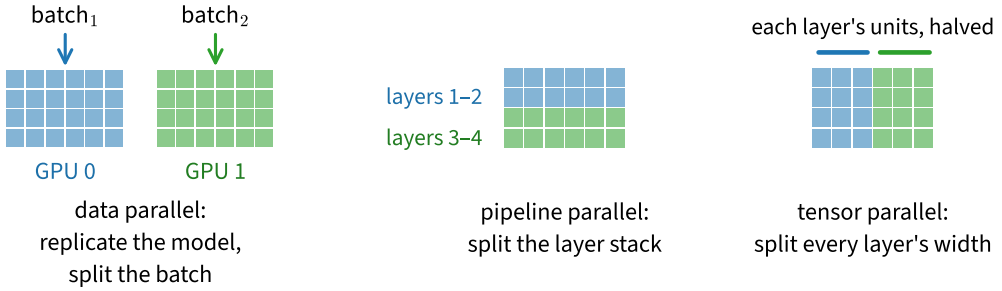


Figure 13.15: Three ways to split a model across two devices, with the cells of each device in its own color: replicate the model and split the batch (data parallel); split the layer stack (pipeline parallel); split every layer's width (tensor parallel).

Data parallelism gives every device a full copy of the model and a different slice of the minibatch. Each device runs the whole network on its slice, the devices sum their gradients, and every copy takes the same optimizer step. It works for any model that fits on one GPU, and the devices communicate once per step. *Pipeline parallelism* splits the stack of layers: each device holds a few consecutive layers and passes its activations to the next device. *Tensor parallelism* splits every layer: each device holds a slice of each weight matrix and computes a slice of each layer's output, so the devices communicate several times per layer. The last two let a model that is too large for one GPU run at all. AlexNet used such a split in 2012:

each of its layers was divided between two GPUs, because the network did not fit into one card's 3 GB (Krizhevsky et al. 2012). Today pipeline and tensor parallelism matter at the scale of the models in the Language Models part, and we leave them to it.

Data parallelism does not let us train a larger model. Every device holds a full copy of the parameters, the gradients, and the optimizer state, which take 16 bytes per parameter for fp32 training with Adam (Section 13.4). A model that does not fit on one GPU therefore does not fit on two. What data parallelism offers is speed: k devices share the work of each minibatch.

13.5.2 Data Parallelism by Hand

A data-parallel step on k devices has four parts (Figure 13.16). We split the minibatch into k equal shards. Each device computes the gradient of the loss on its shard, starting from its own copy of the parameters. The devices then sum their k gradients, so that every device holds the total. Finally, every device applies the same update. Since all copies start equal and receive the same update, they stay equal without ever being compared.

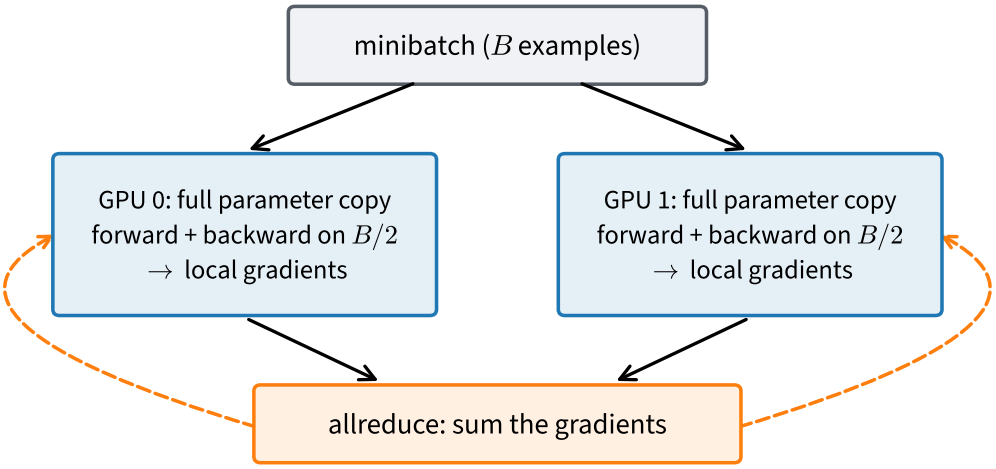


Figure 13.16: One data-parallel step on two GPUs. Split the minibatch, run identical forward and backward passes on each replica, sum the gradients with an allreduce, then apply the identical update everywhere.

We build each part with explicit tensor operations, so that every byte that moves between devices appears in the code.

13.5.2.1 The Model and the Data

We use the LeNet of Section 6.6: two convolutional layers with average pooling, followed by three fully connected layers. We make two of the changes that Chapter 7 makes to it on the way to modern networks: ReLU activations in place of sigmoids, and He initialization (Section 4.4.2.2) in place of Xavier initialization. Instead of a module, we write the network

as a function of a list of parameter tensors. The parameters, and later their gradients, are then plain tensors that we can copy from device to device.

```
def new_params():
    """LeNet's parameters on the CPU: He-initialized weights, zero biases."""
    params = []
    for shape in [(6, 1, 5, 5), (16, 6, 5, 5), (120, 400), (84, 120),
                  (10, 84)]:
        fan_in = math.prod(shape[1:])
        params += [torch.randn(shape) * math.sqrt(2 / fan_in),
                   torch.zeros(shape[0])]
    return params

def lenet(params, X):
    W1, b1, W2, b2, W3, b3, W4, b4, W5, b5 = params
    H = F.avg_pool2d(F.relu(F.conv2d(X, W1, b1, padding=2)), 2)
    H = F.avg_pool2d(F.relu(F.conv2d(H, W2, b2)), 2).flatten(1)
    H = F.relu(F.linear(H, W3, b3))
    H = F.relu(F.linear(H, W4, b4))
    return F.linear(H, W5, b5)

def to_device(split, device):
    """A whole Fashion-MNIST split on `device`: images in [0, 1], labels."""
    return split.data.to(device).unsqueeze(1) / 255, split.targets.to(device)
```

LeNet is small. Its 61,706 parameters in ten tensors take 247 KB in fp32. The 60,000 training images take 188 MB as floats, little enough to keep a copy on every GPU. A copy on every GPU takes the input pipeline out of the measurements below: each device reads its share of a minibatch from its own copy, and the only data that cross between devices are gradients.

```
assert d2l.num_gpus() >= 2, 'This section needs two GPUs'
data = d2l.FashionMNIST()
devices = [d2l.try_gpu(i) for i in range(2)]
train_data = [to_device(data.train, d) for d in devices]  # A copy each
test_data = to_device(data.val, devices[0])
base = new_params()          # One initialization for every run below
print(f'{sum(p.numel() for p in base)} parameters in {len(base)} tensors; '
      f'images {tuple(train_data[0][0].shape)} on each of 2 devices')
```

```
61706 parameters in 10 tensors; images (60000, 1, 28, 28) on each of 2 devices
```

13.5.2.2 Broadcast and Allreduce

Two operations make the step parallel. *Broadcast* copies one device's tensors to every other device; we need it once, to give every device the same initial parameters. Allreduce takes one tensor per device, sums them, and leaves the sum on every device; we need it once per step, for the gradients. The simplest allreduce is a *star*: device 0 collects the tensors of all other devices and adds them up, then sends the sum back.

```
def broadcast(params, device):
    """A copy of every tensor in `params` on `device`, ready for autograd."""
    return [p.to(device, copy=True).requires_grad_() for p in params]

def allreduce(data):
    """Sum the tensors in `data`, one per device, onto every device."""
    for i in range(1, len(data)):    # Star: collect and add on device 0 ...
        data[0] += data[i].to(data[0].device)
    for i in range(1, len(data)):    # ... then copy the sum back
        data[i].copy_(data[0])
```

`.to(device)` copies a tensor from one device to another; the copy and the addition on device 0 are the whole algorithm.

Two vectors, one on each GPU, show what an allreduce leaves behind:

```
x = [torch.full((1, 2), i + 1.0, device=d) for i, d in enumerate(devices)]
print('before:', [x_j.tolist() for x_j in x])
allreduce(x)
print('after: ', [x_j.tolist() for x_j in x])
```

```
before: [[[1.0, 1.0]], [[2.0, 2.0]]]
after:  [[[3.0, 3.0]], [[3.0, 3.0]]]
```

Each device now holds $1 + 2 = 3$ in every entry.

13.5.2.3 The Data-Parallel Step

Each device computes the gradient of the mean loss over its own shard. With k shards of equal size, the average of the k shard gradients is the gradient of the mean loss over the whole minibatch. An allreduce followed by a division by k therefore gives every device the full-batch gradient. (The exercises ask what changes when the shards differ in size.)

LeNet's gradient consists of ten tensors, and we could allreduce them one at a time. Every allreduce has a fixed cost, however few bytes it moves, so we instead concatenate each device's gradients into one flat vector and make a single allreduce per step. Section 13.5.4 measures how much this saves. The training step computes every device's gradient on its shard, sums the gradients with one allreduce, and applies the update. Its argument `communicate=False` skips the allreduce, so that every device updates its own copy from its own shard alone. The result is not a training step, but it lets us time the work that the devices do without communicating.

```
def shard_grads(params, data, idx):
    """Gradient of the mean loss on each device's share of the minibatch
    `idx` (example indices), as one flat vector per device."""
    grads = []
    for ps, (X, y), i in zip(params, data, idx.chunk(len(params))):
        i = i.to(X.device, non_blocking=True)  # Don't wait for the device
        F.cross_entropy(lenet(ps, X[i]), y[i]).backward()
        grads.append(torch.cat([p.grad.flatten() for p in ps]))
        for p in ps:
```

```
            p.grad = None              # The next backward starts from zero
    return grads

def train_step(params, data, idx, lr, communicate=True):
    """One data-parallel SGD step on the minibatch `idx`; without
    communication, every device steps on its own shard's gradient."""
    grads = shard_grads(params, data, idx)
    if communicate:
        allreduce(grads)               # One message per device and step
    with torch.no_grad():
        for ps, g in zip(params, grads):
            for p, g_p in zip(ps, g.split([p.numel() for p in ps])):
                p -= lr / len(params) * g_p.view_as(p)  # Mean of k grads
```

`shard_grads` computes the gradient on every device in turn and flattens it. It then clears `p.grad`, because `backward` adds to the gradients that are already stored. The step runs in a single Python process, which drives all devices in turn: it issues the forward and backward pass for device 0, then for device 1, then the copies of the allreduce. The kernels of different devices can still run at the same time, because each GPU works through its own queue while Python fills the next one.

13.5.2.4 A One-Step Check

A data-parallel step should update the parameters as a step on one device with the same minibatch does. We check this once, before reading any timing. The check is cheap, and it catches errors that an accuracy curve hides. A step that sums the shard gradients instead of averaging them trains with a learning rate k times too large, and it may still reach a plausible accuracy. From one random initialization, we take one step on one GPU and one step on two GPUs, on the same 256 examples.

The check runs in true fp32. Elsewhere in this section the GPU's tensor cores multiply fp32 operands in tf32, with a 10-bit mantissa (Section 13.4.2). A minibatch of 256 and one of 128 can select different convolution algorithms, and their tf32 rounding errors would hide the small difference that we look for.

```
idx = torch.randperm(len(data.train))[:256]
updates = []
torch.set_float32_matmul_precision('highest')  # True fp32 in matmuls ...
with torch.backends.cudnn.flags(enabled=True, allow_tf32=False):  # ... convs
    for k in (1, 2):
        params = [broadcast(base, d) for d in devices[:k]]
        train_step(params, train_data[:k], idx, lr=0.2)
        updates.append([p.detach().cpu() - b
                        for p, b in zip(params[0], base)])
torch.set_float32_matmul_precision('high')
largest = max(u.abs().max() for u in updates[0])
diff = max((u1 - u2).abs().max() for u1, u2 in zip(*updates))
print(f'largest update {largest:.1e}, largest difference {diff:.1e}')
```

```
largest update 7.7e-02, largest difference 3.0e-08
```

The two updates differ by less than a hundred-thousandth of the largest update. That is fp32

rounding, because the two steps add the same numbers in a different order. The parallel step computes what the single-device step computes, so any difference in speed that we measure below comes from how the work is spread over the devices.

13.5.3 Ring Allreduce

A star allreduce routes everything through device 0. For a gradient of N bytes on k devices, device 0 receives $(k-1)N$ bytes and sends $(k-1)N$ bytes, while the links of the other devices carry N bytes each way and then sit idle. The time of the star therefore grows linearly in k. *Ring allreduce* (Patarasuk and Yuan 2009) spreads the traffic evenly over all devices.

Arrange the k devices in a ring in which each sends only to its successor, and split every device's gradient vector into k chunks of N/k bytes. The allreduce runs in two phases (Figure 13.17). In the first phase, called *reduce-scatter*, each device sends one chunk to its successor and adds the chunk it receives from its predecessor to its own copy of that chunk. After $k-1$ such steps, every device owns one chunk that holds the sum over all devices, a different chunk on each device. In the second phase, called *all-gather*, the finished chunks travel around the ring for another $k-1$ steps until every device has all of them. At every step every device sends one chunk and receives one chunk, so each device sends $(k-1)N/k$ bytes per phase, and the two phases together send

$$\frac{2(k-1)}{k}\,N \quad \text{bytes per device.} \tag{13.9}$$

The traffic per device thus does not grow with k: it is N bytes at $k=2$ and approaches $2N$ from below as k grows. For a ResNet-18, with 45 MB of fp32 gradients, on four GPUs each device sends 67 MB, where the hub of a star would send 134 MB and receive as much. The two phases are also useful on their own. Reduce-scatter leaves each device with the sum of one part of the gradient, and all-gather assembles a vector from parts held on different devices. The sharded training of the next section uses them separately.

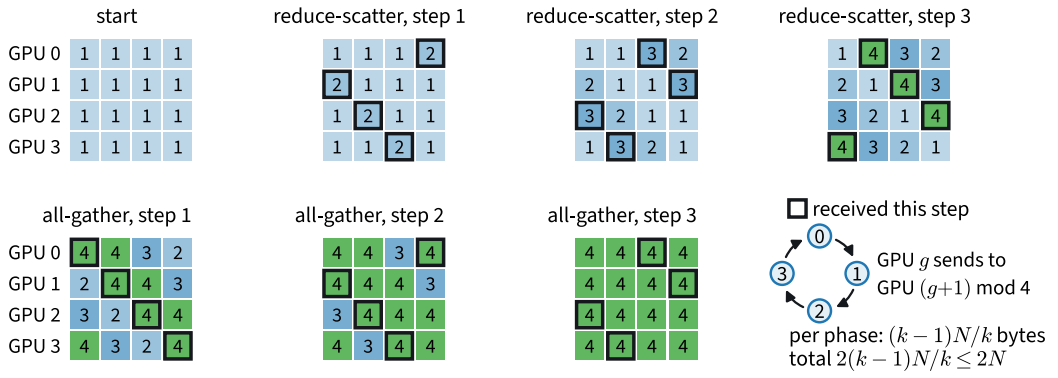


Figure 13.17: Ring allreduce on four GPUs, computed step by step. Reduce-scatter accumulates one chunk per device around the ring; all-gather then distributes the finished chunks. Each cell counts how many of the four contributions a chunk holds, so a chunk holding 4 is complete. An outlined cell is the chunk that a GPU received in that step, and the ring in the last panel shows which GPU sends to which.

The chunk bookkeeping is easy to get wrong, so we check it in a simulation. Each NumPy vector stands for the gradient on one device, and one step of the ring copies every message before any device adds or stores what it receives, since all devices send at once. In reduce-scatter, device j sends chunk $j - s$ in step s, so device j ends with the finished chunk $j + 1$, which it sends first in all-gather.

```
def ring_allreduce(xs):
    """Ring allreduce of the vectors xs, one per device; returns every
    device's result and the bytes that each device sent."""
    k = len(xs)
    chunks = [np.array_split(x.copy(), k) for x in xs]  # chunks[j][c]
    sent = 0
    for s in range(k - 1):  # Reduce-scatter: add the received chunk
        msgs = [chunks[j][(j - s) % k].copy() for j in range(k)]
        for j, msg in enumerate(msgs):
            chunks[(j + 1) % k][(j - s) % k] += msg
        sent += msgs[0].nbytes
    for s in range(k - 1):  # All-gather: replace by the received chunk
        msgs = [chunks[j][(j + 1 - s) % k].copy() for j in range(k)]
        for j, msg in enumerate(msgs):
            chunks[(j + 1) % k][(j + 1 - s) % k] = msg
        sent += msgs[0].nbytes
    return [np.concatenate(c) for c in chunks], sent
```

Four devices with vectors of 1,024 floats, $N = 4{,}096$ bytes each, should all end with the sum and each send $2 \cdot 3/4 \cdot N = 6{,}144$ bytes:

```
rng = np.random.default_rng()
xs = [rng.standard_normal(1024, dtype=np.float32) for _ in range(4)]
results, sent = ring_allreduce(xs)
print(f'every device holds the sum: '
      f'{all(np.allclose(r, sum(xs), atol=1e-5) for r in results)}; '
      f'sent per device: {sent} bytes, '
      f'2(k - 1)N/k = {2 * 3 * xs[0].nbytes // 4} bytes')
```

```
every device holds the sum: True; sent per device: 6144 bytes, 2(k - 1)N/k = 6144 bytes
```

Every device ends with the sum, and each sends exactly the bytes of Equation 13.9.

The byte count does not determine the time on its own. The ring takes $2(k - 1)$ steps one after another, and each step includes the fixed time of a transfer, so the fixed cost of a ring allreduce grows linearly in k while its bytes do not. For small messages, where the fixed cost dominates, collective libraries switch to algorithms with fewer sequential steps, such as trees, whose depth grows only like $\log_2 k$; an exercise derives where the crossover lies. On our machine, the ring also has fewer independent links to spread its traffic over than its picture suggests. Without peer-to-peer access, a transfer from one GPU to its neighbor goes up one PCIe link into host memory and down another (Section 13.2.4.2), so all hops share the host's memory system and two PCIe host bridges. Equation 13.9 says how many bytes each device must move. How fast they move is a property of the machine that we have to measure.

13.5.4 Communication Cost and Scaling

We can now ask the question of the opening: does a second GPU shorten a step? The answer comes from comparing the computation that moves to the new device with the communication that it adds.

13.5.4.1 A Cost Model for One Step

On one device, a step on a minibatch of B examples takes time $t_{\text{compute}}(B)$ for the device's local work: the forward and backward pass and the update. On k devices, each device does this work for B/k examples, and the step also waits for the allreduce of the N gradient bytes that every device holds. A transfer of n bytes takes time $\alpha + n/\beta$, a fixed cost α plus the bytes divided by a bandwidth β (Hockney 1994).

We apply this model to the allreduce as a whole and write its time as $\alpha + 2N/\beta$, which defines β by measurement: $2N/\beta$ is the time of an allreduce so large that its fixed cost does not matter. On two devices, every allreduce makes each device send N bytes and receive N, so β counts the bytes that leave and enter a device together. For our star, whose two copies run one after the other, β is the bandwidth of one copy. The step time on k devices is then approximately

$$t_{\text{step}}(k) \approx t_{\text{compute}}(B/k) + m\,\alpha + \frac{2N}{\beta}, \tag{13.10}$$

where m is the number of allreduce calls per step, each with the fixed cost α. At $k = 1$ both communication terms vanish. Only the first term shrinks as devices are added; the other two stay. A second device therefore shortens the step only if $t_{\text{compute}}(B) - t_{\text{compute}}(B/2)$, the computation it takes over, exceeds $m\alpha + 2N/\beta$, the communication it adds. The three quantities t_{compute}, α, and β can each be measured on their own, and we do so for LeNet.

13.5.4.2 Latency and Bandwidth of the Allreduce

An allreduce of a single number moves almost no bytes, so its time is α. An allreduce of 64M floats, 256 MB per device, takes so long that α hardly matters, and $\beta = 2N/(t - \alpha)$ follows from its time t. We measure both on two GPUs. A training loop does not wait for the devices after each allreduce; the next kernels queue behind it. We therefore time many calls in a row and wait once at the end, since a wait after every call would add its own delay to α.

```
def allreduce_time(n, devices, calls):
    """Seconds per allreduce of n floats per device."""
    x = [torch.zeros(n, device=d) for d in devices]
    return d2l.Benchmark(lambda: allreduce(x), warmup=10 * calls,
                         repeats=3 * calls, devices=devices).time

alpha = allreduce_time(1, devices, calls=100)
n = 2**26                                  # 64M floats ...
N = 4 * n                                  # ... 256 MB
beta = 2 * N / (allreduce_time(n, devices, calls=3) - alpha)
print(f'alpha = {1e6 * alpha:.0f} us, beta = {beta / 1e9:.1f} GB/s')
```

```
alpha = 57 us, beta = 18.9 GB/s
```

`d2l.Benchmark` waits for every device in `devices` before it starts and before it stops the clock. It warms up with ten times as many calls as it times, because the first few hundred allreduces of a run can be slower than the rest, which would inflate α. The star allreduce reaches about 20 GB/s. On two devices it consists of two copies of N bytes, one after the other, so each copy runs at about 20 GB/s as well. The GPUs' copy engines move the bytes over PCIe and through host memory, and this is about what one PCIe 4.0 link carries in one direction after that detour. The fixed cost α is that of two driver-managed copies.

At these bandwidths, moving LeNet's 247 KB of gradients takes $2N/\beta$, at most about a tenth of a millisecond. Does it matter whether they travel as one message or as ten? We allreduce LeNet's ten gradient tensors first one at a time, then all at once as one concatenated vector. The cell prints both times next to the predictions of Equation 13.10:

```
grads = [[torch.zeros_like(p, device=d) for p in base] for d in devices]
flat = [torch.cat([g.flatten() for g in G]) for G in grads]
def separately():
    for g in zip(*grads):              # One allreduce per tensor
        allreduce(list(g))
n_bytes, m = flat[0].nbytes, len(base)
for name, f, calls in (('separately', separately, m),
                       ('at once', lambda: allreduce(flat), 1)):
    t = d2l.Benchmark(f, warmup=1000, repeats=300, devices=devices).time
    print(f'{name:10s}: {1e6 * t:4.0f} us measured, '
          f'{1e6 * (calls * alpha + 2 * n_bytes / beta):4.0f} us predicted')
```

```
separately:  578 us measured,  598 us predicted
at once   :   81 us measured,   83 us predicted
```

Ten calls take several times as long as one call that moves the same bytes.

The two constants predict both times to within a few percent.

For a gradient this small, the number of messages decides the communication time, and the bytes hardly matter. This is why collective libraries pack many small gradient tensors into a few large buffers before they send them, as our training step does by hand.

13.5.4.3 LeNet on One and Two GPUs

A step on two GPUs spends $\alpha + 2N/\beta$ on one allreduce. What it saves is the difference between $t_{\text{compute}}(256)$, the time of one device's local work on a minibatch of 256 examples, and $t_{\text{compute}}(128)$, the time of the same work on half of it. We measure both on one GPU with `communicate=False`. A third measurement runs the local work on two GPUs at once, 128 examples each, which is what the data-parallel step does before its allreduce:

```
params, t_local = [broadcast(base, d) for d in devices], {}
for k, b in ((1, 256), (1, 128), (2, 128)):
    t_local[k, b] = d2l.Benchmark(
        lambda: train_step(params[:k], train_data[:k], idx[:k * b], 0.2,
                           communicate=False),
        warmup=500, repeats=150, devices=devices[:k]).time
```

```
    print(f'{k} GPU(s), {b} examples each: {1e3 * t_local[k, b]:.2f} ms')
```

```
1 GPU(s), 256 examples each: 2.87 ms
1 GPU(s), 128 examples each: 2.83 ms
2 GPU(s), 128 examples each: 5.36 ms
```

On one GPU, halving the minibatch shortens the local work by far less than half. A LeNet layer applied to a few hundred small images is far too little work to occupy an RTX 4090, so each kernel finishes before the next one is ready to start. The step is close to the overhead-bound regime of Section 13.1.3: its time is set mostly by the number of kernels launched rather than by the arithmetic inside them, and halving the minibatch leaves the number of kernels unchanged.

On two GPUs, the local work takes about twice as long as on one GPU with the same 128 examples. One Python thread launches the kernels of both devices, one device after the other, so twice as many launches wait for a single interpreter, and each GPU idles while Python serves the other. Equation 13.10 assumes that the devices work independently; here they share one launcher.

On two GPUs the local work thus takes longer than on one GPU with the whole minibatch, the allreduce adds $\alpha + 2N/\beta$ on top, and Equation 13.10 predicts that a step on two GPUs takes longer than a step on one. The predicted step time is the one-GPU step, plus the difference between the local work on two GPUs and on one, plus $\alpha + 2N/\beta$. To test the prediction, we train for five epochs on one GPU and on two, from the same initialization `base`, with minibatches of 256 examples. Both runs visit the training set in the same five random orders, drawn once, so that they differ only in rounding. The function `train` takes the list of orders, one per epoch. It returns the average time per step over all epochs but the first, which includes start-up costs, and then the test accuracy:

```
def train(params, train_data, test_data, orders, batch_size=256, lr=0.2):
    """Data-parallel SGD on the devices that hold `params` and `train_data`,
    one epoch per example order; returns seconds per step, test accuracy."""
    devices = [X.device for X, _ in train_data]
    times = []
    for order in orders:
        t0 = time.perf_counter()
        for idx in order.split(batch_size):
            train_step(params, train_data, idx, lr)
        for d in devices:  # Wait until every device has finished
            torch.cuda.synchronize(d)
        times.append((time.perf_counter() - t0)
                     / math.ceil(len(order) / batch_size))
    X, y = test_data
    with torch.no_grad():
        acc = (lenet(params[0], X).argmax(1) == y).float().mean().item()
    return sum(times[1:]) / (len(times) - 1), acc
```

```
orders, t_step = [torch.randperm(len(data.train)) for _ in range(5)], {}
for k in (1, 2):
    params = [broadcast(base, d) for d in devices[:k]]
```

```
    t_step[k], acc = train(params, train_data[:k], test_data, orders)
    print(f'{k} GPU(s): {1e3 * t_step[k]:.2f} ms per step, '
          f'test accuracy {acc:.2f}')
predicted = (t_step[1] + t_local[2, 128] - t_local[1, 256]
             + alpha + 2 * n_bytes / beta)
print(f'2 GPUs predicted: {1e3 * predicted:.2f} ms per step')
```

```
1 GPU(s): 2.83 ms per step, test accuracy 0.86
2 GPU(s): 5.64 ms per step, test accuracy 0.87
2 GPUs predicted: 5.40 ms per step
```

The step on two GPUs is slower than the step on one, as the separately measured terms predict. For LeNet, a second GPU makes training slower.

The measured step usually lies within a few percent of the prediction. Both are bound by the one Python thread that drives the two GPUs, so they follow the speed of a single CPU core, and on a machine busy with other work the two can differ by a fifth or more.

Both runs reach the same test accuracy to within a point or two. They start from the same parameters and see the examples in the same order, so they differ only in rounding, which a thousand steps of SGD amplify. Two runs on one GPU differ about as much, because a GPU does not always add the same numbers in the same order.

LeNet is too small for data parallelism. A second device shortens a step when the computation on each shard is long compared with $m\alpha + 2N/\beta$, and that requires two things. The per-device batch must be large enough to keep each GPU busy, so that $t_{\text{compute}}(B/k)$ is close to $t_{\text{compute}}(B)/k$. And the model must perform many operations per byte of gradient, so that the computation it moves outweighs the exchange it adds. LeNet performs 0.4 million multiply-adds per image, under two per byte of its gradient. A ResNet-18 on Fashion-MNIST images enlarged to 64×64 pixels performs 2.2 billion multiply-adds per image with 45 MB of gradients, about fifty per byte.

13.5.4.4 Strong and Weak Scaling

Every reported speedup holds something fixed, and two conventions are common. Equation 13.10 holds the global minibatch B fixed while k grows: the same work, finished sooner if all goes well. This is *strong scaling*, and it is what we measured for LeNet. The alternative holds the per-device batch fixed, so that the global minibatch grows to kB_{device}: more work per unit of time. This is *weak scaling*, and its speedups are quoted as throughput, examples per second. Weak scaling is the easier test, because t_{compute} per device does not shrink and only the communication has to stay small next to it. A larger global minibatch, however, changes the optimization itself (Section 9.8), so a gain in examples per second under weak scaling is not a gain of the same size in the time it takes to reach a given loss.

We can now predict from three measurements whether more GPUs shorten a step, and we have seen why they do not for LeNet. Our hand-written version also leaves room for improvement. It waits for the whole backward pass before it communicates, although the gradients of the last layers are ready long before those of the first.

It also drives every device from one Python thread, and its star routes all traffic through device 0 instead of a ring.

Section 13.6 replaces our version with the frameworks' production tools and measures them on a model large enough to gain from more GPUs.

13.5.5 Summary

Data parallelism trains a copy of the model on every device, each on its share of the minibatch. The copies stay equal because an allreduce sums their gradients before every update, and a one-step check confirms that the parallel step computes what one device computes. A ring allreduce, a reduce-scatter followed by an all-gather, moves $2(k-1)N/k$ bytes per device, at most $2N$ however many devices take part. Whether another device shortens a step follows from $t_{\text{step}}(k) \approx t_{\text{compute}}(B/k) + m\alpha + 2N/\beta$, whose terms can each be measured. For LeNet, the fixed cost α of an allreduce outweighs its bytes, which is why gradients travel as one message. Halving LeNet's minibatch does not halve its computation either, so a second GPU makes each step slower. Data parallelism shortens steps for models whose per-device computation is long compared with the exchange of their gradients. A reported speedup must say whether it holds the global or the per-device batch fixed.

13.5.6 Exercises

1. **Data versus pipeline parallelism.** Consider a stack of L fully connected layers of width d, with fp32 weights and no biases, trained on two GPUs with minibatch size B. Data parallelism allreduces the gradients of all weights. Two-stage pipeline parallelism (Figure 13.15) places the first $L/2$ layers on one GPU and the rest on the other, so the forward pass sends the $B \times d$ activations at the cut to the second GPU and the backward pass returns their gradient.
    1. Count the bytes that cross between the two GPUs per step in each scheme, and evaluate both for $L = 32$, $d = 4096$, and $B = 256$. For which relation between B, L, and d does data parallelism move fewer bytes?
    2. In the pipeline, each GPU idles while the other computes. To shorten the idle time, split the minibatch into m micro-batches (Huang et al. 2019). In the forward phase, the first GPU hands each micro-batch to the second as soon as it is done, so that the second GPU works on micro-batch j while the first works on micro-batch $j+1$. The backward phase starts after the last forward pass and runs the same schedule from the second GPU to the first. Assume that every micro-batch takes the same time on each stage. Derive the fraction of the step during which each GPU idles, and evaluate it for $m = 1$ and $m = 8$.
2. **Gradient normalization.** Two recipes compute a data-parallel gradient from k shards of a minibatch of B examples. In recipe A, each device sums its per-example losses, the devices sum their gradients, and the update divides by B. In recipe B, which our training step uses, each device averages its per-example losses, and the devices average their gradients.
    1. Show that both recipes yield the gradient of the full-batch mean loss when every shard holds B/k examples.
    2. A language model averages its loss over tokens. Two shards with the same number of sequences hold 3000 and 1000 tokens. Find the weight that recipe B gives a token on each shard, relative to the weight the token has in the full-batch token mean.
    3. Modify recipe B so that it is exact for any shard sizes, and identify the extra quantity that the devices must communicate before the update.

3. **Strong-scaling speedup.** Model the compute time of a shard of s examples as $t_{\text{compute}}(s) = t_0 + c\,s$, where t_0 does not shrink with the shard. Let each device send b bytes per second over its link, while it receives at the same rate. This per-direction rate b is not the β of Equation 13.10, which counts the bytes in both directions. By Equation 13.9, one step at a fixed global batch B takes

$$t_{\text{step}}(k) = t_0 + \frac{c\,B}{k} + \frac{2(k-1)}{k}\,\frac{N}{b},$$

with no communication at $k = 1$. The strong-scaling speedup is $S(k) = t_{\text{step}}(1)/t_{\text{step}}(k)$.
   1. Show that $S(k) > 1$ holds for one $k > 1$ exactly when it holds for every $k > 1$, and find the condition. Find the limit of $S(k)$ as $k \to \infty$.
   2. A model with 11.2 million fp32 parameters has $t_0 = 2$ ms and $c = 0.2$ ms per example, and it is trained with $B = 256$. Compute $S(2)$, $S(4)$, and the limit for $b = 20$ GB/s and for $b = 5$ GB/s. Find the rate b below which no number of GPUs speeds up this model.
   3. Sending the gradients in bf16 instead of fp32 halves N. For $b = 5$ GB/s, recompute $S(4)$ and the limit of $S(k)$, and state which of the two gains more from the halving.
4. **Parameter servers and the ring.** Before collective libraries became standard, distributed training often summed gradients through *parameter servers* (Li, Andersen, et al. 2014): s server machines each hold a slice of N/s bytes of the parameters. In every step, each of k workers pushes its gradient for each slice to that slice's server, the server adds up the k contributions and updates its slice, and every worker pulls the updated slices back.
   1. Count the bytes that each worker and each server send and receive per step. Find the number of servers at which a server's link is no busier than a worker's.
   2. Now let the k workers act as the servers, each holding $1/k$ of the parameters. Show that the traffic of each device equals Equation 13.9, and match the push and pull phases to the two phases of ring allreduce.
5. code **Ring allreduce by hand.** ⊠ This exercise needs four GPUs. The star pattern routes all traffic through device 0, while the ring of Figure 13.17 spreads it over all k devices.
   1. For k devices holding N bytes each, count the bytes that the star and the ring move in total, and the bytes that the busiest device sends and receives. The star starts its broadcast only after its sum is complete. Predict the ratio of their times under two assumptions. In the first, each device has one link that carries b bytes per second in each direction at the same time, and all links work concurrently. In the second, the machine runs one transfer at a time at b bytes per second.
   2. Write `ring_allreduce(data)` with the interface of `allreduce`: split each tensor into k chunks with `chunk`, and implement reduce-scatter and all-gather with explicit `.to()` copies. Check it against `allreduce` for $k \in \{2, 3, 4\}$, guarding with `d2l.num_gpus()`.
   3. Time both functions with `d2l.Benchmark` on 64M floats per device at $k = 2$ and $k = 4$, and decide which assumption of the first part describes your machine.
6. code **Gradients in bf16.** ⊠ Sending gradients in bf16 halves their bytes but rounds them. In a ring, each reduce-scatter hop adds the incoming partial sum to the local chunk, so a bf16 ring rounds the partial sum to bf16 once per hop.
   1. At the parameters returned by `new_params`, compute the gradients of `lenet` on k disjoint Fashion-MNIST shards of 128 examples each, and flatten each shard's

gradients into one vector. Add the vectors one after another, rounding to bf16 after every addition, and measure the relative error of the result against an fp32 sum for $k \in \{2, 8, 64\}$. Repeat with each vector rounded to bf16 once and the sum accumulated in fp32. Explain how each error depends on k.

2. Compare the error at $k = 8$ with the relative difference between the gradients of two shards, and decide whether the rounding matters for SGD.

7. **Latency in the allreduce.** ▣ The byte count of Equation 13.9 ignores the fixed cost of starting a transfer. Let one transfer of n bytes over a link take $\alpha + n/b$, where b is the rate of one link in one direction.
   1. Derive the time t_{ring} of a ring allreduce of N bytes on k devices, and describe how each of its two terms behaves as k grows.
   2. A binary-tree allreduce sums up the tree and broadcasts back down. It takes $2 \log_2 k$ steps, each of which sends the full N bytes, so

$$t_{\text{tree}} = 2 \log_2 k \left(\alpha + \frac{N}{b} \right). \tag{13.11}$$

   For $k = 8$, $\alpha = 10$ µs, and $b = 20$ GB/s, find the message size below which the tree is faster than the ring.
   3. The allreduce of Equation 13.10 takes $\alpha + 2N/\beta$. The gradient size $N_{1/2}$ at which it spends half of its time on the fixed cost is called its half-performance length. Express $N_{1/2}$ in α and β, and compute it from the values that you measured in Section 13.5.4. How many copies of LeNet's gradient would one message have to carry to reach it? Production data-parallel libraries pack gradients into buffers of tens of megabytes (25 MB by default in PyTorch's `DistributedDataParallel`); judge that size against your $N_{1/2}$.

8. [extended] **Where data parallelism helps.** ▣ LeNet is too small for a second GPU to help. Widen it, timing `train_step` with `communicate=False` on one and on two GPUs to measure $t_{\text{compute}}(b)$, and find out which terms of Equation 13.10 the width changes.
   1. Generalize `new_params` and `lenet` so that the channel counts of both convolutions and the widths of the two hidden dense layers scale with a factor w. Compute the gradient bytes $N(w)$ and the forward-pass multiply-adds per example, and show how their ratio depends on w.
   2. For $w \in \{1, 2, 4, 8, 16\}$, measure the time per step of `train` at $k = 1$ and $k = 2$ with global batch 256, and plot the speedup against w.
   3. Overlay the speedup that Equation 13.10 predicts from your measured $t_{\text{compute}}(128)$ and $t_{\text{compute}}(256)$ and from the α and β measured in the allreduce experiment of Section 13.5.4. Find the speedup that the prediction approaches as w grows. Find the achieved compute rate in FLOP/s below which the second GPU helps at large w.
   4. Repeat the prediction and the measurement at $w = 16$ with global batch 1024, and explain why the batch size raises the limit of the previous part while the width does not.

13.6 Multi-GPU in Practice

In Section 13.5 we built data parallelism by hand, and a second GPU made the training steps of LeNet slower: each GPU had too little work to outweigh even a small exchange of gradients. A larger model gives each GPU more work for every byte it exchanges, so it should

fare better. How much better, once the exchange is organized well? And what can several GPUs do for a model that is too large for any one of them?

A ResNet-18 on Fashion-MNIST images enlarged to 64×64 pixels gives each GPU far more work than LeNet. It performs 2.2 billion multiply-adds per image, about fifty for every byte of its 45 MB of gradients, so the work that a second GPU takes over is large compared with the gradients that the two GPUs must exchange. Whether this advantage survives depends on how the exchange is organized. The hand-written loop of Section 13.5 waited for the whole backward pass before it communicated, although the gradients of the last layers were ready long before those of the first.

It also issued the work of every GPU from one Python thread and routed all gradients through device 0.

Data parallelism also leaves one cost untouched: every GPU keeps a full copy of the training state. The GPT of Section 13.4 with 36 blocks instead of 18 has 1.9 billion parameters. Its weights, their gradients, and Adam's moment estimates take 29 GiB, more than the 24 GiB of an RTX 4090. Data parallelism would place these 29 GiB on every one of our four GPUs, so four GPUs cannot train this model any more than one can.

We first replace the hand-written loop with each framework's production tool for data parallelism, which starts communicating while the backward pass is still running. We measure how ResNet-18 then scales on two and four GPUs, and compare the measurement with the cost model of Section 13.5. We then divide the training state among the GPUs instead of copying it, which reduces the memory per GPU but adds communication. Section 13.7 applies both to a transformer.

```
import json
import os
import random
import sys
```

13.6.1 Production Data Parallelism

Production data parallelism computes what the loop of Section 13.5.2 computes. Every GPU holds a copy of the model, computes the gradient of the loss on its share of each minibatch, and applies the average of the gradients of all shares. What changes is how the work is organized, and above all when the communication starts.

PyTorch changes two more things. Every GPU gets a process of its own, and NCCL replaces our star with the ring of Section 13.5.3, or with a tree for small messages, so that no device carries the traffic of all the others.

13.6.1.1 Overlapping Communication with the Backward Pass

The backward pass computes gradients from the last layer to the first, so the gradients of the last layers are complete long before those of the first. Our loop in Section 13.5.2 nevertheless waited for the whole backward pass before it started the allreduce, and the links between the GPUs sat idle in the meantime. The production tools group the gradients into *buckets* of a few tens of megabytes, in the order in which the backward pass produces them, and start the allreduce of a bucket as soon as its last gradient exists (S. Li et al. 2020). While one bucket travels between the GPUs, the GPUs compute the gradients of the next (Figure 13.18).

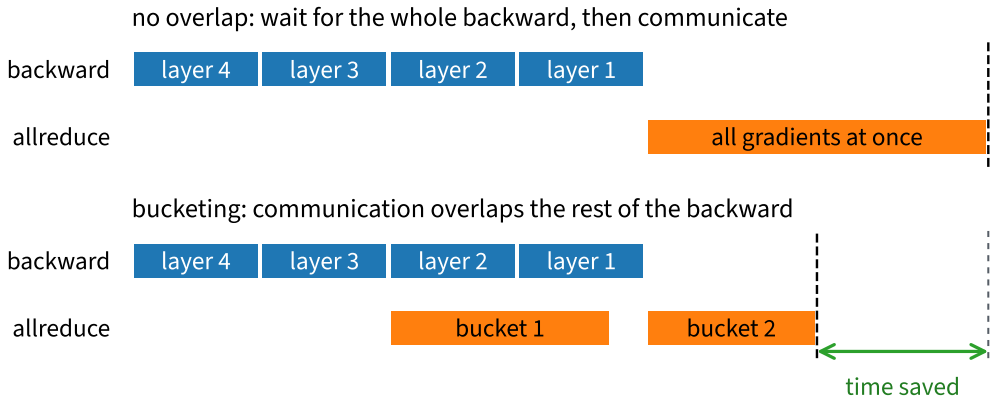


Figure 13.18: Top: the allreduce waits for the whole backward pass. Bottom: the gradients are grouped into buckets, and each bucket's allreduce starts as soon as the backward pass has produced it. Only the last bucket runs after the backward pass has ended.

The size of a bucket is a compromise between the two parts of an allreduce's time, a fixed cost α per call and the bytes divided by the bandwidth β. Small buckets start early but incur α many times. A single bucket for all gradients incurs α once, but it waits for the whole backward pass. Whatever the size, the last bucket holds the gradients of the first layers, which the backward pass computes last. Its allreduce has nothing left to overlap with, so its time adds to every step. Overlap therefore shortens the communication of a step, but it cannot remove it.

In PyTorch, `DistributedDataParallel` (DDP) implements bucketing. It registers a hook on every parameter that marks the parameter's gradient as ready, and it starts the allreduce of a bucket once all of the bucket's gradients are ready. A bucket holds up to 25 MiB (the argument `bucket_cap_mb`), and the buckets are filled in the reverse order of the model's parameters, which is close to the order of the backward pass. ResNet-18's 45 MB of gradients form three buckets of about 9, 26, and 9 MB. The last of them holds the gradients of the first two thirds of the layers, which have few channels and therefore few weights. The allreduces run on a CUDA stream of their own, so they proceed while the backward kernels run.

13.6.1.2 The Execution Model

In Section 13.5 one Python thread issued the kernels of every GPU, one GPU after the other. The time that the host spends on a step then grows with the number of GPUs, and for a model with many small kernels the host becomes the bottleneck, as it did for LeNet. PyTorch instead starts one process per GPU. Every process runs an ordinary training script on its own GPU, so no GPU waits for the host to serve the others. The older `nn.DataParallel` drives all GPUs from one process, as our loop did, and PyTorch's documentation recommends DDP even within one machine.

The processes find each other through `torch.distributed.init_process_group`, which connects them through NCCL, NVIDIA's library of collective operations, and DDP sends its allreduces through NCCL. The launcher `torchrun` starts k copies of a script and tells each copy its *rank*, its index from 0 to $k-1$, through environment variables. A notebook is a single process, so we write each script as a string and start it with the following helper. It collects the

lines of JSON that the processes print, process by process in the order of their ranks. Every script begins with `init_distributed`: it pins its process to the GPU that `torchrun` assigned to it and joins the process group.

```
import glob
import subprocess
import tempfile

def torchrun(script, nproc, *args, env=None):
    """Run Python source in nproc processes; return their JSON lines."""
    with tempfile.TemporaryDirectory() as tmp:
        path = os.path.join(tmp, 'script.py')
        with open(path, 'w') as f:
            f.write(script)
        launcher = os.path.join(os.path.dirname(sys.executable), 'torchrun')
        # Each process prints to a file of its own (--redirects 1), so that
        # the lines of different processes cannot interleave
        run = subprocess.run(
            [launcher, '--standalone', f'--nproc-per-node={nproc}',
             '--log-dir', tmp, '--redirects', '1', path, *map(str, args)],
            capture_output=True, text=True, env={**os.environ, **(env or {})})
        if run.returncode != 0:
            raise RuntimeError(run.stderr[-3000:])
        logs = sorted(glob.glob(f'{tmp}/*/attempt_0/*/stdout.log'),
                      key=lambda log: int(log.split(os.sep)[-2]))  # By rank
        return [json.loads(line) for log in logs for line in open(log)
                if line.startswith('{')]

def init_distributed():
    """Join the process group of torchrun; return the rank, the number of
    processes, and the GPU of this process."""
    device = torch.device('cuda', int(os.environ['LOCAL_RANK']))
    torch.cuda.set_device(device)
    torch.distributed.init_process_group('nccl', device_id=device)
    return (torch.distributed.get_rank(), torch.distributed.get_world_size(),
            device)
```

13.6.1.3 The Training Step

Our model is the ResNet-18 of Section 7.4 with a stem for small images: a 3×3 convolution without max-pooling, followed by four stages of two residual blocks each, with He initialization as in Section 7.4. We save it, since later chapters train it on several GPUs.

```
def resnet18(num_classes, in_channels=1):
    """ResNet-18 for small images: a 3x3 stem without max-pooling, He
    initialization."""
    def stage(num_channels, first=False):
        return [d2l.Residual(num_channels, use_1x1conv=not first,
                             strides=1 if first else 2),
                d2l.Residual(num_channels)]
    net = nn.Sequential(
```

```
        # No stem bias: the batch normalization after the conv removes it
        nn.Conv2d(in_channels, 64, kernel_size=3, padding=1, bias=False),
        nn.BatchNorm2d(64), nn.ReLU(),
        *stage(64, first=True), *stage(128), *stage(256), *stage(512),
        nn.AdaptiveAvgPool2d(1), nn.Flatten(), nn.Linear(512, num_classes))
    with torch.no_grad():  # Create the lazy parameters of the blocks
        net.eval()(torch.zeros(1, in_channels, 32, 32))
    for m in net.modules():  # He initialization, zero biases
        if isinstance(m, (nn.Conv2d, nn.Linear)):
            nn.init.kaiming_normal_(m.weight, nonlinearity='relu')
            if m.bias is not None:
                nn.init.zeros_(m.bias)
    return net.train()
```

As in Section 13.5, every GPU keeps its own copy of the training set, here enlarged once to 64×64 pixels, which takes 983 MB in fp32. The input pipeline then stays out of our measurements, and the only data that cross between GPUs are gradients. We also count the gradient bytes N that every step must sum across the GPUs:

```
d2l.FashionMNIST()                  # Download once, before any process runs
num_params = d2l.num_params(resnet18(10))
N = 4 * num_params                  # Gradient bytes in fp32
print(f'{num_params / 1e6:.1f}M parameters, N = {N / 1e6:.0f} MB')
```

```
11.2M parameters, N = 45 MB
```

The training script follows. Two parts make it data-parallel: `init_distributed`, which joins the process to the others, and the `DDP` wrapper around the model. The rest is a single-GPU training loop. Every process takes its own slice of each global minibatch of kb examples. All processes draw the same random permutation, because they share a seed, and the process of rank r takes the r-th of the k slices of b examples. For data that stay in host memory, `DistributedSampler` performs the same split for a `DataLoader`. The script times its steps with `d2l.Benchmark` (Section 13.1) and prints the seconds per step.

```
TRAIN_DDP = r'''
import json, sys, torch
import torch.distributed as dist
from torch.nn import functional as F
from torch.nn.parallel import DistributedDataParallel as DDP
from d2l import torch as d2l

rank, k, device = d2l.init_distributed()
torch.backends.cudnn.benchmark = True  # Tune convolutions for fixed shapes
b, num_steps, seed = map(int, sys.argv[1:])  # b examples per GPU and step
torch.manual_seed(seed)                # The same permutations everywhere

data = d2l.FashionMNIST().train        # A copy on every GPU, at 64 x 64
X = F.interpolate(data.data.to(device)[:, None] / 255, 64, mode='bilinear')
y = data.targets.to(device)
model = DDP(d2l.resnet18(10).to(device), device_ids=[device])
opt = torch.optim.SGD(model.parameters(), lr=0.1)
```

```
def minibatches():
    """This process's b examples of every global minibatch of k * b."""
    while True:
        perm = torch.randperm(len(y), device=device)
        for s in range(len(y) // (k * b)):
            yield perm[(s * k + rank) * b:][:b]

def train_step(i):
    opt.zero_grad(set_to_none=True)
    F.cross_entropy(model(X[i]), y[i]).backward()
    opt.step()

batches = minibatches()
t = d2l.Benchmark(lambda: train_step(next(batches)), warmup=10,
                  repeats=num_steps).time
print(json.dumps({'rank': rank, 'step_time': t}))
dist.destroy_process_group()
'''
```

13.6.1.4 A One-Step Check

Before timing anything, we check, as in Section 13.5.2, that a step on two GPUs computes what we expect. We compare it with steps on one GPU, from the same initialization and on the same 256 examples, in full fp32 arithmetic so that rounding differences stay small. The layer to watch is batch normalization (Section 7.3). It normalizes every channel with the mean and the variance over the minibatch, and when the minibatch is split across GPUs, these statistics can cover either one GPU's shard or the whole minibatch.

The script below takes one DDP step on k GPUs, each GPU on its half of 256 random images. Process 0 also keeps an unwrapped copy of the model and takes two reference steps on its GPU: one on all 256 examples at once, and one that accumulates the gradients of the same two halves before it updates the parameters.

```
CHECK_DDP = r'''
import copy, json, sys, torch
import torch.distributed as dist
from torch.nn import functional as F
from torch.nn.parallel import DistributedDataParallel as DDP
from d2l import torch as d2l

rank, k, device = d2l.init_distributed()
torch.backends.cudnn.allow_tf32 = False  # Full fp32 convolutions
torch.manual_seed(int(sys.argv[1]))      # The same model and data everywhere
X = torch.rand(256, 1, 64, 64, device=device)
y = torch.randint(0, 10, (256,), device=device)
model = d2l.resnet18(10).to(device)
ref = copy.deepcopy(model)               # An unwrapped copy for one GPU

def update(net, shards):
    """Parameter change of one SGD step on the mean loss over the shards."""
```

```
    before = [p.detach().clone() for p in net.parameters()]
    for X_j, y_j in shards:
        (F.cross_entropy(net(X_j), y_j) / len(shards)).backward()
    with torch.no_grad():
        for p, b in zip(net.parameters(), before):
            p -= 0.1 * p.grad
    return [p.detach() - b for p, b in zip(net.parameters(), before)]

ddp = update(DDP(model, device_ids=[device]),
             [(X.chunk(k)[rank], y.chunk(k)[rank])])
if rank == 0:
    gap = lambda u: max((a - b).abs().max().item() for a, b in zip(ddp, u))
    halves = update(copy.deepcopy(ref), list(zip(X.chunk(k), y.chunk(k))))
    print(json.dumps({'largest': max(u.abs().max().item() for u in ddp),
                      'full': gap(update(ref, [(X, y)])),
                      'halves': gap(halves)}))
dist.destroy_process_group()
'''
if d2l.num_gpus() >= 2:
    r = d2l.torchrun(CHECK_DDP, 2, random.getrandbits(32))[0]
    print(f"largest update {r['largest']:.1e}\n"
          f"largest difference from one GPU on all 256: {r['full']:.1e}\n"
          f"largest difference from one GPU on two accumulated halves: "
          f"{r['halves']:.1e}")
```

```
largest update 1.6e-02
largest difference from one GPU on all 256: 8.0e-04
largest difference from one GPU on two accumulated halves: 1.2e-07
```

The DDP step agrees with the accumulated halves to rounding error, well under one percent of the largest update, but it differs from the step on all 256 examples by more than one percent of the largest update. DDP normalizes each GPU's shard with the statistics of that shard, so it computes what one GPU computes when it accumulates the gradients of the same shards (Section 13.4.4). The mean and the variance over 128 examples are already close to those over the whole minibatch. Over a few examples per GPU they become noisy, and `nn.SyncBatchNorm` then computes them over the whole minibatch, which adds collectives to every batch normalization layer (exercise 7).

13.6.2 Scaling on Four GPUs

We measure both conventions of Section 13.5.4: weak scaling, which holds the minibatch per GPU fixed while the number of GPUs k grows and reports examples per second, and strong scaling, which holds the global minibatch fixed and reports how much sooner a step finishes. The function `step_time(k, b)` returns the seconds per step on k GPUs with b examples per GPU, averaged over 50 steps after a warm-up.

```
def step_time(k, b, num_steps=50):
    """Seconds per DDP step on k GPUs with b examples per GPU."""
    ranks = d2l.torchrun(TRAIN_DDP, k, b, num_steps, random.getrandbits(32))
    return max(r['step_time'] for r in ranks)
```

Weak scaling first, with 256 examples per GPU:

```
ks = [k for k in (1, 2, 4) if k <= d2l.num_gpus()]
weak = {k: step_time(k, 256) for k in ks}
for k in ks:
    print(f'{k} GPU(s): {k * 256 / weak[k]:5.0f} examples/s, speedup '
          f'{k * weak[1] / weak[k]:.2f}, efficiency {weak[1] / weak[k]:.0%}')
```

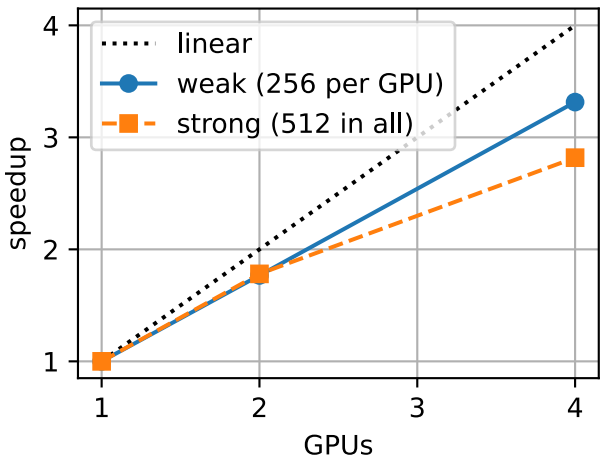
```
1 GPU(s):  2282 examples/s, speedup 1.00, efficiency 100%
2 GPU(s):  4033 examples/s, speedup 1.77, efficiency 88%
4 GPU(s):  7565 examples/s, speedup 3.31, efficiency 83%
```

Two GPUs process about 1.7 times as many examples per second as one, and four GPUs a little over three times as many. The *efficiency*, the speedup divided by k, falls gradually as GPUs are added. Under weak scaling every GPU does the same work at every k. The efficiency is therefore the ratio of the single-GPU step time to the step time on k GPUs, and what it loses is the communication that the step does not hide.

Strong scaling holds the global minibatch at 512, so each of the k GPUs processes $512/k$ examples while the allreduce keeps its size:

```
strong = {k: step_time(k, 512 // k) for k in ks}
for k in ks:
    print(f'{k} GPU(s), {512 // k} examples each: speedup '
          f'{strong[1] / strong[k]:.2f}, efficiency '
          f'{strong[1] / strong[k] / k:.0%}')
d2l.plot(ks, [ks, [k * weak[1] / weak[k] for k in ks],
              [strong[1] / strong[k] for k in ks]], 'GPUs', 'speedup',
         legend=['linear', 'weak (256 per GPU)', 'strong (512 in all)'],
         fmts=(':', '-o', '--s'))
```

```
1 GPU(s), 512 examples each: speedup 1.00, efficiency 100%
2 GPU(s), 256 examples each: speedup 1.78, efficiency 89%
4 GPU(s), 128 examples each: speedup 2.82, efficiency 70%
```



At two GPUs the two conventions give about the same speedup, because 256 and 512 examples keep an RTX 4090 about equally busy. At four GPUs they diverge. Strong scaling leaves

each GPU 128 examples per step, so the step's computation shrinks while its communication does not, and the efficiency falls well below that of weak scaling. The same mechanism made LeNet slower on two GPUs; here its effect is much weaker.

13.6.2.1 Predicted and Measured Communication

The cost model Equation 13.10 of Section 13.5.4 predicts the time of a step on k GPUs from the compute time of a shard of B/k examples, the number m of allreduce calls with fixed cost α each, and the gradient bytes N at bandwidth β:

$$t_{\text{step}}(k) \approx t_{\text{compute}}(B/k) + m\,\alpha + \frac{2N}{\beta}.$$

Under weak scaling the first term does not change with k, so the difference $t_{\text{step}}(2) - t_{\text{step}}(1)$ is the communication that a step exposes. For 45 MB of gradients in a few calls, the bandwidth term dominates. The model counts the whole allreduce, so $2N/\beta$ is the exposed time if nothing overlaps. Overlap as in Figure 13.18 lowers the exposed time: only the allreduce of the last bucket remains exposed, provided that the earlier buckets finish while the backward pass still runs.

As in Section 13.5.4, we define β as $2N$ divided by the time of a large allreduce on two GPUs, so β counts the bytes that leave and enter a GPU together. We measure it with a bare allreduce of 256 MB per GPU, the same size as there, through the library that the training uses:

`ALLREDUCE` times `torch.distributed.all_reduce` through NCCL in every process of a two-process launch.

```
ALLREDUCE = r'''
import json, torch
import torch.distributed as dist
from d2l import torch as d2l

_, _, device = d2l.init_distributed()
x = torch.zeros(2**26, device=device)   # 64M floats: 256 MB
t = d2l.Benchmark(lambda: dist.all_reduce(x), repeats=10).time
print(json.dumps({'beta': 2 * x.nbytes / t}))
dist.destroy_process_group()
'''
def allreduce_beta(env=None):
    """Bandwidth of a bare 256 MB allreduce on two GPUs, in bytes/s."""
    return min(r['beta'] for r in d2l.torchrun(ALLREDUCE, 2, env=env))
```

```
if d2l.num_gpus() >= 2:
    beta = allreduce_beta()
    print(f'beta = {beta / 1e9:.1f} GB/s; k=2: predicted 2N/beta = '
          f'{1e3 * 2 * N / beta:.0f} ms, measured exposed '
          f'{1e3 * (weak[2] - weak[1]):.0f} ms per step')
```

```
beta = 4.9 GB/s; k=2: predicted 2N/beta = 18 ms, measured exposed 15 ms per step
```

The measured exposed time is close to $2N/\beta$, the time of an allreduce that nothing overlaps. Overlap hides far less of the communication on this machine than the bucket model predicts.

DDP's last bucket holds 8.7 of the 45 MB of gradients, so with full overlap a step would expose a fifth of $2N/\beta$, about 4 ms.

The bucket model makes three assumptions: every allreduce starts as soon as its bucket is complete, it runs as fast as the bare benchmark, and the backward kernels beside it run as fast as they do alone. A trace of one step, which records when every kernel starts and ends, tests them (exercise 4). On this machine the third fails: the kernels of an allreduce take a share of the GPU's processors and memory bandwidth, and the backward kernels beside them run more slowly.

In a trace of the step on two GPUs, the backward kernels take about a tenth longer than in the same step without communication, and the allreduces take more than twice as long as the bare benchmark. Together they lengthen the step by about 15 ms.

On a machine whose GPUs share an NVLink domain, each GPU exchanges data with the others at 450 GB/s in each direction on the H100 (NVLink 4) and at 900 GB/s on the B200 (NVLink 5), hundreds of times the rate measured here (Section 13.2.4.2). The allreduce of ResNet-18's gradients then takes well under a millisecond, and the cost model predicts near-linear scaling for this model with or without overlap.

13.6.2.2 The Collective Transport

At $k = 2$ each GPU sends and receives at $\beta/2$ in each direction, about 2.5 GB/s. A copy between two of our GPUs, staged through host memory, can run at up to 23 GB/s (Section 13.2.4.2), about nine times as fast. The cause lies in how NCCL moves data between GPUs that lack peer-to-peer access. It falls back to buffers in host memory, and by default GPU kernels copy the data into and out of these buffers with ordinary loads and stores across PCIe. The copy engines, the units that perform a GPU's memory copies, move the same bytes faster. An undocumented NCCL parameter, `NCCL_SHM_USE_CUDA_MEMCPY=1`, makes NCCL use them. It exists in the NCCL versions that our pinned frameworks bundle, and NCCL 2.31 removed it. We measure the bare allreduce with the parameter set:

```
if d2l.num_gpus() >= 2:
    beta_ce = allreduce_beta({'NCCL_SHM_USE_CUDA_MEMCPY': '1'})
    print(f'beta = {beta / 1e9:.1f} GB/s by default, '
          f'{beta_ce / 1e9:.1f} GB/s with the copy engines '
          f'({beta_ce / beta:.1f}x)')
```

```
beta = 4.9 GB/s by default, 23.0 GB/s with the copy engines (4.7x)
```

The bandwidth rises nearly five-fold, and with it the predicted communication time of ResNet-18 falls from about 18 ms to about 4 ms.

PyTorch 2.11 bundles NCCL 2.28.9. On our machine, DDP training with the parameter set deadlocks within seconds, although the same collectives complete in the bare benchmark. Exercise 5 reproduces both outcomes under a time limit. We therefore train with NCCL's default.

One environment variable changed β nearly five-fold here. Whether such a setting is safe has to be checked on the training run itself, not only on a benchmark of bare collectives.

13.6.3 Sharding the Training State

Data parallelism makes training faster, but it makes no room for a larger model: every GPU holds all of the weights, the gradients, and the optimizer state. With Adam in fp32 these take 16 bytes per parameter (Section 13.4.1), so a 24 GiB card holds at most 1.6 billion parameters before it stores a single activation. The 36-block GPT from the start of this section has 1.9 billion parameters. At 16 bytes each they take 29 GiB, which no single GPU of ours holds, and data parallelism would need these 29 GiB on every GPU.

13.6.3.1 Partitioning the Optimizer State, Gradients, and Weights

Each GPU needs a full copy of the weights only while it computes with them, and it never needs the full gradient or the full optimizer state. The ZeRO family of methods (Rajbhandari et al. 2020) removes the redundant copies in three stages. Stage 1 partitions the optimizer state: each GPU keeps Adam's moment estimates for $1/k$ of the parameters and updates only those parameters. Stage 2 also partitions the gradients, and stage 3 also partitions the weights. Table 13.5 lists what each GPU then holds.

Table 13.5: Training state per GPU with Adam in fp32 on k GPUs, and for the GPT with 36 blocks and 1.9 billion parameters on $k = 4$.

each GPU holds	weights	gradients	Adam state	bytes per parameter	GPT, $k = 4$
data parallelism	all	all	all	16	28.6 GiB
ZeRO stage 1	all	all	$1/k$	$8 + 8/k$	17.9 GiB
ZeRO stage 2	all	$1/k$	$1/k$	$4 + 12/k$	12.5 GiB
ZeRO stage 3 (FSDP)	$1/k$	$1/k$	$1/k$	$16/k$	7.1 GiB

The collectives follow from the identity of Section 13.5.3: an allreduce is a reduce-scatter followed by an all-gather. In stages 1 and 2 the reduce-scatter leaves each GPU with the summed gradient of the parameters it owns, which is all that its optimizer needs. Each GPU then updates its $1/k$ of the parameters, and an all-gather assembles the updated parameters on every GPU. The two phases move the same bytes as the allreduce of data parallelism, now separated by the update. Stages 1 and 2 therefore reduce the memory per GPU at no cost in communication. For our GPT, stage 1 on four GPUs already brings the training state below the 24 GiB of one card.

Stage 3 is known as *fully sharded data parallelism* (FSDP) (Y. Zhao et al. 2023). Each GPU keeps only its shard of every block's weights, and an all-gather assembles a block's weights just before the block computes and releases them right after (Figure 13.19). The backward pass gathers each block's weights once more, releases them after the block's backward pass, and reduce-scatters the block's gradients. A step thus performs two all-gathers and one reduce-scatter of N bytes, three phases where an allreduce has two, so FSDP communicates 1.5 times as much as data parallelism. No full copy of a block lives longer than the block's computation, so the memory per GPU is $16/k$ bytes per parameter plus the activations and one or two gathered blocks.

Table 13.6 lists the four collectives that data-parallel and sharded training use.

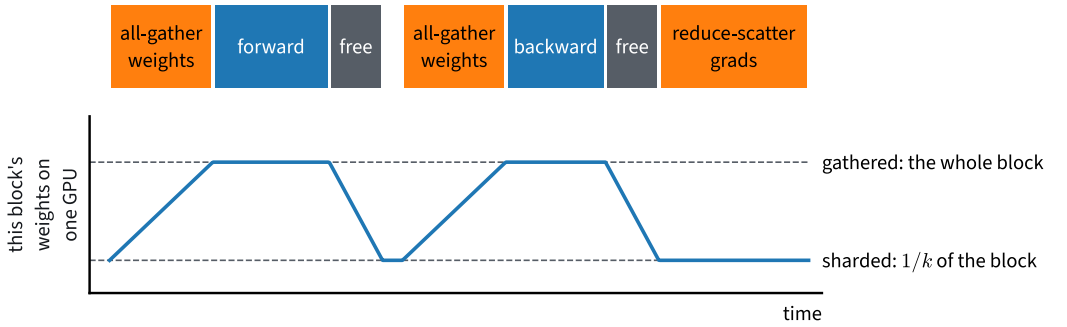


Figure 13.19: The life of one block's weights under FSDP on one GPU. Between uses the block is sharded, and each GPU holds $1/k$ of it; the other blocks stay sharded throughout. An all-gather assembles the whole block for its forward pass, after which it is released. A second all-gather assembles it for the backward pass, after which it is released again, and a reduce-scatter leaves each GPU with its share of the gradients.

Table 13.6: The collective operations behind data-parallel and sharded training.

collective	what every GPU ends with	where it appears
allreduce	the full sum	gradients in data parallelism
reduce-scatter	one part of the sum	gradients in ZeRO and FSDP
all-gather	every part, concatenated	weights in ZeRO and FSDP
all-to-all	a different part from each GPU	tokens routed to the experts of a mixture of experts (Section 11.6)

In PyTorch, `fully_shard` from `torch.distributed.fsdp` applies FSDP to a module. It is called once for every transformer block and once for the whole model, so that every block becomes a unit that is gathered and released on its own. The older `FullyShardedDataParallel` wrapper still works, and PyTorch's documentation recommends `fully_shard` for new code. Stage 1 alone is available as `torch.distributed.optim.ZeroRedundancyOptimizer`. DDP itself needs more than the first row of Table 13.5: its buckets hold a second copy of the gradients, 4 more bytes per parameter, unless the argument `gradient_as_bucket_view=True` makes the gradients views into the buckets.

13.6.3.2 Memory and Communication

Sharding reduces memory and adds communication, and the cost model says how long the communication takes. For our GPT, the fp32 gradients take N = 7.7 GB, four bytes per parameter. On k GPUs, a ring allreduce makes every GPU send $2(k-1)N/k$ bytes and receive as many (Section 13.5.3), $1.5N$ in each direction at $k = 4$ against N at $k = 2$. If every GPU keeps the bandwidth β of about 5 GB/s that we measured on two, one allreduce of data parallelism takes $3N/\beta$, about 5 s, and the three phases of FSDP take $4.5N/\beta$, about 7 s.

The computation of a step is much shorter. The forward and the backward pass take about

six floating-point operations per parameter and token (Section 11.7). Two sequences of 1,024 tokens per GPU, which fit next to the 7.1 GiB of stage 3 with the activation checkpointing of Section 13.4.3, take about $6 \cdot 1.9 \cdot 10^9 \cdot 2048 \approx 2.4 \times 10^{13}$ operations. At the RTX 4090's full tf32 rate of about 80 TFLOP/s that is 0.3 s, and 0.4 s with the forward pass that checkpointing repeats. On our machine, sharding lets the model train on four GPUs, at a few percent of the speed that its computation would allow. Over NVLink 5, at 900 GB/s in each direction, the same three phases take about 20 ms, and FSDP is the standard way to train models whose state does not fit on one GPU.

The arithmetic also says when to shard. A model whose training state fits on one GPU next to its activations trains fastest with data parallelism. On a 24 GiB card with Adam in fp32 that limit lies at about a billion parameters, fewer for long sequences. The 18-block GPT of Section 13.4 sits at this limit. Stage 3 would shrink its 15.1 GiB per GPU to 3.8 GiB at $k = 4$ and free 11 GiB for activations, but on our machine the extra communication would take far more time than the larger micro-batches save.

Past the limit, ZeRO stages 1 and 2 reduce the memory at no cost in communication, and stage 3 removes the remaining copies with 1.5 times the communication. Splitting the weight matrices of every layer across GPUs instead, *tensor parallelism* (exercise 8), and the forms of parallelism that span several machines belong to the Language Models part.

We can now train a model on all GPUs of one machine with the tools that practitioners use, predict from the cost model what a step will cost, and check the prediction against a measurement. Our measurements also show how much depends on the connection between the GPUs: when they exchange data through host memory, communication sets the pace long before the arithmetic does. Section 13.7 applies the method of this chapter to a transformer, where precision, compilation, and data parallelism interact.

13.6.4 Summary

Production data parallelism computes what the hand-written loop computes, but the framework writes the collectives, and the allreduce of each bucket of gradients starts while the backward pass is still running. A one-step check against a single GPU shows what the data-parallel step computes, including how batch normalization treats a split minibatch. On our four RTX 4090s ResNet-18 gains from every added GPU, but its exposed communication is close to the cost model's time for an allreduce that nothing overlaps: on this machine, overlap hides little of it. ZeRO and FSDP divide the 16 bytes per parameter of the training state among the GPUs, stages 1 and 2 at no cost in communication and stage 3 at 1.5 times the communication of data parallelism.

13.6.5 Exercises

1. code **Splitting the data.** In `TRAIN_DDP` every process takes its own slice of each global minibatch: all processes draw the same permutation, and the process of rank r takes the slice that starts at $(sk + r)b$ in step s.
    1. Remove the rank from this offset, so that every process takes the same b examples. Predict what `step_time` reports at $k = 2$ relative to the correct script, and which global minibatch the averaged gradient corresponds to. Then check, with each process printing a checksum of its first minibatch.
    2. Restore the offset and give each process a seed of its own. Which property of the split is lost, and does it matter for training?

3. Name a check that catches the first mistake and is cheap enough to run at every launch.

2. **Weak-scaling extrapolation.** Suppose that a step on k GPUs takes the single-GPU step time t_1 plus an exposed communication time that is proportional to the per-GPU ring traffic of Section 13.5.3, $2(k-1)N/k$.
   1. Express the weak-scaling efficiency $E(k)$ through a single constant, fit the constant to your measurement at $k = 2$, and predict $E(4)$. Compare your prediction with the measured $E(4)$ and with the prediction of a model in which the exposed time is the same for every $k \geq 2$.
   2. Predict $E(8)$ for an eight-GPU machine of the same design as ours, on which every transfer between GPUs is staged through host memory (Section 13.2.4.2). How does the total ring traffic through host memory grow with k, and in which direction should this move a measured $E(8)$ relative to your prediction?
   3. With gradient accumulation (Section 13.4.4), each GPU runs a micro-batches of 256 examples per optimizer step and communicates only after the last one. Starting from your measured $E(4)$, find the smallest a that raises the efficiency at $k = 4$ to 95%. What happens to the global minibatch, and why does that matter (Section 9.8)?

3. **Cost of sharding.** ▯ Count communication per GPU and step in units of N, the bytes of one full set of fp32 gradients.
   1. A variant of ZeRO stage 3 keeps every gathered block from its forward pass until its backward pass. Which collective does this remove, and how much of the memory saving of stage 3 does it give up, in bytes per parameter of Table 13.5?
   2. Mixed-precision FSDP gathers the weights in bf16 and reduce-scatters the gradients in bf16, while every GPU keeps its shard of the fp32 weights and of Adam's state. Compare its bytes per step with those of data parallelism in fp32, and its memory per parameter with the last row of Table 13.5.
   3. For the GPT with 36 blocks and 1.9 billion parameters on our four GPUs, and with the β that you measured, find the number of tokens per GPU and step at which the computation, 6 operations per parameter and token at 80 TFLOP/s, takes as long as the communication of stage 3. How many sequences of 1,024 tokens is that, and do their activations fit next to the training state (Section 13.4.1)? How does gradient accumulation change the answer when FSDP gathers the weights anew for every micro-batch?

4. code **Bucket size.** ▯ Model the overlap of Figure 13.18 as follows. The backward pass takes time t_b and releases the gradients in n buckets of equal size, bucket i at time $i\,t_b/n$. The allreduces of the buckets run one at a time and in order, and each takes $\alpha + \tau/n$, where $\tau = 2N/\beta$ is the communication time of all gradients and α is a fixed cost per collective. The exposed communication is the time from the end of the backward pass to the end of the last allreduce.
   1. Show that the exposed communication is

$$t_{\text{exposed}}(n) = \max\Big(\alpha + \frac{\tau}{n},\ n\alpha + \tau - \frac{n-1}{n}\,t_b\Big). \tag{13.12}$$

   2. Find the n that minimizes t_{exposed} when τ is well below t_b, and again when $\tau > t_b$. What does a bucket that is too small cost, and what does one that is too large cost?

3. Pass `bucket_cap_mb` to `DDP` in `TRAIN_DDP`, sweep it from 1 MB to 100 MB at $k = 2$ with `step_time`, and plot the exposed communication against the bucket size. Estimate t_b as two thirds of the single-GPU step time and τ as $2N/\beta$. Is the measured optimum consistent with your model, and which α does it imply?
4. At the default buckets, the exposed communication is close to τ rather than to the time of the last bucket. Record a few steps at $k = 2$ with `torch.profiler.profile`, once as they are and once with the backward pass inside `model.no_sync()`, which skips the allreduces. Compare the time from the first to the last backward kernel in the two runs, and find when each allreduce kernel starts and ends. Which assumption of the model fails on our machine?

5. code **Transport configuration.** ▣ `allreduce_beta` compares NCCL's default transport with the copy engines that `NCCL_SHM_USE_CUDA_MEMCPY=1` selects, at a single payload of 256 MB. The parameter affects only the transport through host memory that NCCL uses between GPUs that lack peer-to-peer access.
   1. Launch `ALLREDUCE` once with `NCCL_DEBUG=INFO` and find the transport that NCCL reports for each channel. Predict whether the parameter can change anything on your machine.
   2. Extend the comparison to payloads from 1 MB to 256 MB and plot the bandwidth of both transports against the payload. Where does each saturate, and does the five-fold gap hold at every size?
   3. Launch `TRAIN_DDP` on two GPUs with the parameter set (the `env` argument of `d2l.torchrun`), once with the default buckets and once with `bucket_cap_mb` large enough that all gradients form a single bucket, each launch under a time limit that also terminates the processes. Does the deadlock depend on the overlap of communication with the backward pass, and what does the outcome tell you about which workloads the bare benchmark validates?

6. [extended] **Sharded optimizer state.** ▣ Implement ZeRO stage 1 (Rajbhandari et al. 2020) by hand and measure what it saves. Start from `TRAIN_DDP` with `torch.optim.Adam` in place of SGD.
   1. Assign every parameter tensor to one owner process, so that the processes own nearly equal numbers of values, and give each process an Adam optimizer over its own tensors only. After the backward pass and the owners' steps, send each tensor from its owner to all processes with `dist.broadcast`. Check that after a few steps the parameters match those of DDP with a full Adam from the same initialization, and compare with `torch.distributed.optim.ZeroRedundancyOptimizer`.
   2. Read `torch.cuda.memory_allocated` after initialization, before the first optimizer step, and after it, with and without sharding at $k = 2$ and $k = 4$, and account for the differences with Table 13.5; then read `torch.cuda.max_memory_allocated` after the second step and compare its change under sharding with the change in live memory.
   3. Measure the step time with and without sharding. DDP's allreduce hands every process the full summed gradient, of which stage 1 uses only the part that the process owns. Count the bytes per GPU that the allreduce and the broadcasts of your implementation send together, and compare them with the reduce-scatter and all-gather of stage 1 in Section 13.6.3.

Adapted from Stanford CS336, Assignment 2 (Systems), Problems optimizer_state_sharding and optimizer_state_sharding_accounting.

7. code **Synchronized batch normalization.** ⊠ The one-step check of Section 13.6.1 shows that DDP normalizes each GPU's shard with the statistics of that shard.
    1. Convert the model in `CHECK_DDP` with `nn.SyncBatchNorm.convert_sync_batchnorm` after creating its parameters and before wrapping it in `DDP`, and repeat the check. Which of the two references does the DDP step now match?
    2. How many collectives per step does the conversion add to `resnet18`? Measure `step_time` with the converted model at $k = 2$ and $k = 4$ under weak scaling, and compare the added time with the time that the added bytes take at bandwidth β. Which properties of these collectives, other than their size, set their cost?
8. code **Tensor versus data parallelism.** ⊠ Consider the MLP block $\mathbf{Y} = \phi(\mathbf{X}\mathbf{W}_1)\mathbf{W}_2$ with $\mathbf{X} \in \mathbb{R}^{n\times d}$ holding the n tokens of one step, $\mathbf{W}_1 \in \mathbb{R}^{d\times 4d}$, $\mathbf{W}_2 \in \mathbb{R}^{4d\times d}$, and an elementwise activation ϕ, on k devices. Tensor parallelism (Figure 13.15) splits $\mathbf{W}_1$ by columns and $\mathbf{W}_2$ by rows (Shoeybi et al. 2019).
    1. Show that each device can compute, without communication, an $n \times d$ partial result from its columns of $\mathbf{W}_1$ and its rows of $\mathbf{W}_2$, and that $\mathbf{Y}$ is the sum of the k partial results. Which collective forms that sum, and which collective does the backward pass need for the gradient with respect to $\mathbf{X}$?
    2. Data parallelism on the same k devices allreduces the gradients of $\mathbf{W}_1$ and $\mathbf{W}_2$ once per step. Count the values that each scheme sends per device and step, and find the number of tokens n below which tensor parallelism sends less.
    3. Tensor parallelism is in practice confined to the GPUs of one NVLink domain even where it sends less. Explain why, using when each scheme's communication happens relative to the computation that needs its result.
    4. In a script launched with `d2l.torchrun`, apply `parallelize_module` to `nn.Sequential(nn.Linear(d, 4 * d), nn.GELU(), nn.Linear(4 * d, d))` on `init_device_mesh("cuda", (k,))`, with `ColwiseParallel()` for submodule `"0"` and `RowwiseParallel()` for submodule `"2"`. Count the collectives of the forward and the backward pass with `CommDebugMode` from `torch.distributed.tensor.debug`, once for an input with `requires_grad=True` and once without, and explain the difference.

Adapted from Stanford CS336, Assignment 2 (Systems), Problems data_parallel_calcs and tp_calcs.

13.7 Case Study: Optimizing a Transformer

The earlier sections of this chapter took the causes of a slow training step one at a time and found the change that removes each: host overhead, the memory traffic of small operations, wide number formats, and the limits of a single GPU. They also showed that recomputing activations saves memory, at the cost of time, when a step does not fit. A real model suffers from several of these causes at once, and removing one changes how much the others matter. How much faster does its training step become when we apply all the changes in turn?

We answer the question for the GPT of Section 11.2, which has 19 million parameters when we double its width to 512. One training step on 64 windows of 128 bytes performs about $6 \times 1.9 \times 10^7 \times 8{,}192 \approx 9 \times 10^{11}$ floating-point operations: two per parameter and token in the forward pass and four in the backward pass. The tensor cores of an RTX 4090 execute 1.65×10^{14} of them per second in bf16, so the arithmetic of a step needs less than 6 ms.

A plain training step, in fp32 with an AdamW update, takes 25 to 30 ms. Three quarters or more of the card's capacity go unused, and every training run takes several times longer than its arithmetic requires.

We follow the method of Section 13.1. We measure the step, find the resource that limits it, change one thing, check that the model still computes the same function, and measure again. Each change is measured on top of the previous ones, because each moves the bottleneck and so changes what the next one can achieve. For data parallelism we predict the gain from a cost model before we measure it, and at the end a short training run checks that the fastest step still trains the model.

```
from torch.utils.checkpoint import checkpoint
```

```
torch.set_float32_matmul_precision('high')  # fp32 products on tf32 cores
```

13.7.1 Experimental Setup

Every change in this section is judged by the same measurement, so the model, the data, and the way we measure throughput stay fixed throughout.

13.7.1.1 Model and Data

We train `d2l.GPT` as Section 11.2 defines it: six pre-norm blocks with eight attention heads each, rotary position embeddings, RMSNorm and SwiGLU, and an input embedding shared with the output layer. The data are also those of Section 11.2: windows of 128 bytes of WikiText-103 at random offsets, with the 256 byte values as the vocabulary (`d2l.WikiText103`, Section 9.5).

We change one setting: the width, from 256 to 512. The reason is arithmetic intensity, the number of FLOPs that an operation performs per byte it moves (Section 13.1). A product of an $n \times d$ matrix of activations with a $d \times d'$ weight matrix performs $2ndd'$ FLOPs and moves $b(nd + dd' + nd')$ bytes, with b bytes per number. For the $n = 8{,}192$ tokens of a step at width 512, the products of a block reach between 250 and 370 FLOPs per byte in bf16. The ridge point of the RTX 4090, the intensity above which an operation is limited by arithmetic rather than by memory bandwidth, lies at 164 FLOPs per byte. The products of the wider GPT are therefore limited by arithmetic, so a faster arithmetic format shortens them directly. At width 256 the same products lie around the ridge point; an exercise studies that model.

The training text has 270 million bytes, one per token, so it fits on the GPU as an array of 270 MB. The next cell places it there and defines `windows`, which draws every minibatch on the GPU from random offsets; no step then waits for data from the host. The cell also builds the model.

```
device = d2l.try_gpu()
data = d2l.WikiText103(batch_size=64, num_steps=128)
corpus = torch.tensor(data.train, device=device)  # One byte per token

def windows(corpus, batch_size, num_steps=128):
```

```
    """Endless minibatches of windows at random offsets, each position with
    the next byte as its target."""
    offsets = torch.arange(num_steps + 1, device=corpus.device)
    while True:
        starts = torch.randint(len(corpus) - num_steps, (batch_size, 1),
                               device=corpus.device)
        W = corpus[starts + offsets].long()
        yield W[:, :-1], W[:, 1:]

model = d2l.GPT(len(data.vocab), num_hiddens=512).to(device)
num_params = d2l.num_params(model)
print(f'{num_params / 1e6:.1f}M parameters, '
      f'{len(corpus) / 1e6:.0f}M training bytes on the GPU')
```

```
19.0M parameters, 270M training bytes on the GPU
```

The model has 19 million parameters, nearly all of them in the weight matrices of its six blocks.

13.7.1.2 Throughput Measurement

We measure speed as training throughput, in tokens per second, end to end: the time of a step includes drawing the minibatch, the forward and backward passes, and the optimizer update. Every measurement in this section follows the same rules.

setting	value
hardware	one RTX 4090 of the four in our machine; data parallelism uses two and four
model	`d2l.GPT`: 6 blocks, width 512, 8 heads, byte vocabulary, context 128
data	windows of WikiText-103 at random offsets, drawn on the GPU, new for every step
optimizer	AdamW with learning rate 10^{-3} and weight decay 0.01
precision	fp32 with matrix products on the tf32 tensor cores, until bf16 is introduced
timing	`d2l.Benchmark`: 20 warm-up steps, then 50 steps between two device synchronizations
result	tokens per second, and its ratio to the previous configuration

`tokens_per_second` applies these rules with `d2l.Benchmark` of Section 13.1, whose warm-up calls absorb one-time costs such as compilation and memory allocation. `record` stores each result and prints its ratio to the previous one.

```
def tokens_per_second(step, batches, warmup=20, repeats=50):
    """Training throughput of `step` on minibatches drawn from `batches`."""
    X, _ = next(batches)
    seconds = d2l.Benchmark(lambda: step(*next(batches)), warmup,
```

```
                            repeats).time
    return math.prod(X.shape) / seconds

results = {}
def record(name, speed):
    """Store a throughput and print it with its ratio to the previous one."""
    previous = list(results.values())[-1] if results else speed
    results[name] = speed
    print(f'{name}: {speed:,.0f} tokens/s ({speed / previous:.2f}x)')
    return speed
```

One timed window of 50 steps varies by a few percent from run to run, so we read ratios to one decimal place.

13.7.2 A Single GPU

We first make the step fast on one GPU, one change at a time, starting from the step as Section 11.2 wrote it.

13.7.2.1 Baseline

The baseline is the training step of Section 11.2 with plain AdamW: a forward pass, the cross-entropy loss, a backward pass, and the update.

The step runs eagerly. Python issues every operation of the forward pass and autograd every operation of the backward pass, each as one or more kernel launches. The optimizer, created with `fused=True`, updates all weights and moment estimates in a few kernels instead of several per weight tensor. `make_step` packages the step as a function of a minibatch; its arguments `forward` and `bf16` bring in the changes of the next subsections. The notebook runs with the environment variable `TEARDOWN_CUPTI=1`, which detaches the profiler of the next cells from the GPU when a profile ends; without it, every later kernel launch of the process stays slower.

```
def make_step(model, forward=None, bf16=False):
    """An AdamW training step of `model` on a minibatch (X, Y); `forward`
    replaces the model's forward pass, `bf16` turns on autocast."""
    forward = model if forward is None else forward
    optimizer = torch.optim.AdamW(model.parameters(), lr=1e-3,
                                  weight_decay=0.01, fused=True)
    def step(X, Y):
        with torch.autocast('cuda', torch.bfloat16, enabled=bf16):
            loss = F.cross_entropy(forward(X).flatten(0, 1), Y.flatten())
        optimizer.zero_grad(set_to_none=True)
        loss.backward()
        optimizer.step()
        return loss
    return step

step = make_step(model)
speed = record('baseline', tokens_per_second(step, windows(corpus, 64)))
```

```
baseline: 327,847 tokens/s (1.00x)
```

A profile shows where the time of the step goes. `profile` records five steps with `d2l.kernel_times` and prints them with `d2l.show_profile`, both from Section 13.1.4: the time during which the device runs kernels against the wall-clock time of a step, which follows from the measured throughput, the share of the matrix products, counting fused attention kernels among them, and the kernels that take the most time.

```
def profile(step, batches, speed):
    """Kernel profile of `step` on minibatches drawn from `batches`, at a
    measured throughput of `speed` tokens per second."""
    X, _ = next(batches)
    kernels = d2l.kernel_times(lambda: step(*next(batches)), steps=5)
    d2l.show_profile(kernels, 1e3 * math.prod(X.shape) / speed, rows=4)

profile(step, windows(corpus, 64), speed)
```

```
wall clock 24.99 ms per step, device busy 24.44 ms (98%), matrix products 64%
  3.28 ms  13%  Kernel2<cutlass_80_tensorop_s1688gemm_128x128_32x3_n
  3.09 ms  13%  Kernel2<cutlass_80_tensorop_s1688gemm_128x128_32x3_t
  2.84 ms  12%  Kernel2<cutlass_80_tensorop_s1688gemm_128x128_32x3_n
  2.41 ms  10%  vectorized_elementwise_kernel<4, BinaryFunctor<float
```

The device runs kernels for nearly the whole step, so the host keeps up with it. Matrix products take a little under two thirds of the device time. The rest is spread over many short kernels: the elementwise operations of SwiGLU, of the rotary embeddings, and of the residual connections, and the normalizations. Each of them reads its operands from device memory and writes its result back, so together they are limited by memory bandwidth (Section 13.3.2).

13.7.2.2 Host Overhead and Small Kernels

A profile separates two regimes of Section 13.1. Gaps in which the device waits for the host mark the overhead-bound regime: the step runs at the pace of the Python interpreter, and a faster GPU would not shorten it. Device time spread over many short elementwise kernels marks the bandwidth-bound regime: each kernel moves its operands through device memory for a few FLOPs. The remedies differ. Overhead calls for less work on the host per step, and short kernels call for fusion, which merges chains of them into one kernel that reads its inputs once and writes its result once.

Our step is in the second regime: the device hardly waits, but much of its time goes to short kernels. `torch.compile` (Section 13.3) captures the forward pass of the model as a graph and generates kernels for it and for its backward pass, each of which fuses a chain of elementwise operations. It thus replaces several hundred operations that Python and autograd dispatch one at a time by far fewer kernels. The matrix products remain calls to the library, and the optimizer update stays as it is. With `mode='reduce-overhead'`, the compiled kernels of the forward and of the backward pass are recorded as two CUDA graphs and replayed with one launch each (Section 13.3.4), so the host no longer issues them one by one. With `dynamic=False`, every minibatch shape gets kernels specialized to it. The first call with a new shape compiles, which takes seconds, and the warm-up of `tokens_per_second` absorbs it. Before timing anything, we check that the compiled model computes the same logits as the original one, measured by their relative difference.

```
compiled = torch.compile(model, dynamic=False, mode='reduce-overhead')
X, _ = next(windows(corpus, 64))
high = model(X)
error = ((compiled(X) - high).norm() / high.norm()).item()
print(f'relative difference of the compiled logits: {error:.1e}')
assert error < 0.01
step = make_step(model, compiled)
speed = record('+ compile', tokens_per_second(step, windows(corpus, 64)))
```

```
relative difference of the compiled logits: 6.6e-06
+ compile: 368,795 tokens/s (1.12x)
```

The logits agree to three or more significant digits. They are not identical, because the fused fp32 kernels round in a different order than the separate ones, and the tf32 products carry the difference forward. Throughput rises by about a tenth.

A second profile shows where the bottleneck has moved:

```
profile(step, windows(corpus, 64), speed)
```

```
wall clock 22.21 ms per step, device busy 22.25 ms (100%), matrix products 71%
  4.00 ms  18%  Kernel2<cutlass_80_tensorop_s1688gemm_128x128_32x3_n
  3.05 ms  14%  Kernel2<cutlass_80_tensorop_s1688gemm_128x128_32x3_t
  2.83 ms  13%  Kernel2<cutlass_80_tensorop_s1688gemm_128x128_32x3_n
  1.44 ms   6%  fmha_cutlassB_f32_aligned_64x64_k64_sm80
```

The device is now busy for nearly the whole step, and the matrix products take the largest share of its time. The step is limited by the device, mostly by the arithmetic of the matrix products.

Compared with the first profile, the device time per step has fallen and the gaps between kernels have closed. The matrix products take as long as before, while the elementwise kernels have merged into fewer generated ones, whose names begin with `triton_`.

13.7.2.3 Reduced Precision

When the device is busy and the matrix products take most of its time, the next change should make the products faster. In bf16 the tensor cores of the RTX 4090 run at 165 TFLOP/s, twice their tf32 rate, and every activation that is stored or read takes half the bytes (Section 13.4). We apply the mixed-precision recipe of Section 13.4.2: the matrix products of the forward and the backward pass, those of the attention included, run in bf16, and so do the elementwise operations between them, such as SwiGLU. The weights, their gradients, and Adam's moment estimates stay in fp32, and so do the normalizations, the loss, and the residual stream that connects the blocks. The stream is kept in fp32 because each block adds to it an update that can be much smaller than the stream itself, and bf16, with 8 significant bits, would round away an update a few hundred times smaller than the entry it is added to. Since bf16 has the exponent range of fp32, the loss needs no scaling. The check before the timing compares the bf16 logits with the fp32 logits of the same weights; they should agree to the precision of bf16, two to three significant digits.

`torch.autocast` applies this recipe operation by operation (Section 5.5): it runs the matrix products and the attention in bf16 and computes reductions such as the softmax of the loss

in fp32. The embedding and the normalizations receive no cast, so the stream stays in fp32. The backward pass follows the dtypes of the forward pass.

```
X, _ = next(windows(corpus, 64))
with torch.autocast('cuda', torch.bfloat16):
    low = compiled(X).float()
high = compiled(X)
error = ((low - high).norm() / high.norm()).item()
print(f'relative difference of the bf16 logits: {error:.1e}')
assert error < 0.05
step = make_step(model, compiled, bf16=True)
speed = record('+ bf16', tokens_per_second(step, windows(corpus, 64)))
```

```
relative difference of the bf16 logits: 1.7e-03
+ bf16: 644,090 tokens/s (1.75x)
```

The logits agree to within one percent.

Throughput rises by about 1.7 times. That is less than the factor of two of the tensor cores, because not every part of the step halves its work: the normalizations and the residual stream still move fp32 values, the casts between the two formats add traffic of their own, and the optimizer update still reads and writes fp32 weights and moment estimates.

13.7.2.4 Larger Minibatches

Some costs of a step do not grow with the minibatch. The optimizer update reads and writes every weight and both moment estimates once per step, however many tokens the step processes, and a kernel launch costs the same for 64 windows as for 512. A larger minibatch divides these fixed costs among more tokens, and its larger matrix products keep the tensor cores busier. The activations grow in proportion to the minibatch, so the step needs more memory. We raise the minibatch from 64 to 512 windows, or 65,536 tokens per step, and record the memory that the step needs beyond the weights and the optimizer state.

`torch.cuda.max_memory_allocated` reports the most memory allocated since the last `reset_peak_memory_stats`; we subtract what was allocated before the steps (Section 13.4.1.4).

```
torch.cuda.reset_peak_memory_stats()
start = torch.cuda.memory_allocated()
speed = record('+ batch 512', tokens_per_second(step, windows(corpus, 512)))
memory = (torch.cuda.max_memory_allocated() - start) / 2**30
print(f'memory beyond weights and optimizer state: {memory:.1f} GiB')
```

```
+ batch 512: 653,408 tokens/s (1.01x)
memory beyond weights and optimizer state: 8.6 GiB
```

The step needs well under half of the card's 24 GiB.

Throughput stays the same. With the update fused and the kernels replayed as CUDA graphs, almost nothing of the step at batch 64 was a fixed cost for the larger minibatch to divide.

An exercise separates the fixed cost of a step from its cost per token by measuring several minibatch sizes. A third profile shows what limits the step at batch 512:

```
profile(step, windows(corpus, 512), speed)
```

```
wall clock 100.30 ms per step, device busy 100.32 ms (100%), matrix products 57%
 11.62 ms  12%  ampere_bf16_s1688gemm_bf16_128x128_ldg8_f2f_stages_3
 10.77 ms  11%  ampere_bf16_s16816gemm_bf16_128x64_ldg8_f2f_tn
  7.45 ms   7%  Kernel2<cutlass_80_tensorop_bf16_s16816gemm_relu_bf1
  5.62 ms   6%  triton_poi_fused__unsafe_view_mm_mul_silu_silu_backw
```

The device is busy for the whole step, and the matrix products, those inside the fused attention included, take half or more of its time. The largest single kernels are matrix products. Most of the rest goes to the kernels between them, for the normalizations, SwiGLU, the rotary embeddings, and the casts between fp32 and bf16, which move bytes through device memory and leave the tensor cores idle.

Matrix products take a little over half of the device time. The kernels that Inductor generated take most of the rest; the largest of them computes the backward pass of SwiGLU.

13.7.2.5 Activation Checkpointing

Activation checkpointing (Section 13.4.3) reduces the memory of a step by recomputing activations: each block keeps only its input during the forward pass and recomputes everything else during the backward pass. The step at batch 512 uses less than half of the card's memory, so memory does not limit it, and our method predicts that checkpointing can only slow the step down. We measure it to check the prediction, and to see how much memory it would free for a model that needed it.

`Checkpointed` wraps a module so that it runs under `checkpoint` from `torch.utils.checkpoint`. We give a copy of our model, with its weights, checkpointed blocks; the rest of its forward pass stays as `d2l.GPT` defines it. Before the timing we check that it computes the same logits.

```
class Checkpointed(nn.Module):
    """A module that keeps only its input for the backward pass."""
    def __init__(self, module):
        super().__init__()
        self.module = module

    def forward(self, X, **kwargs):
        return checkpoint(self.module, X, use_reentrant=False, **kwargs)

remat = d2l.GPT(len(data.vocab), num_hiddens=512).to(device)
remat.load_state_dict(model.state_dict())
remat.blks = nn.ModuleList(Checkpointed(blk) for blk in remat.blks)
assert torch.allclose(remat(X), model(X), atol=1e-6)
step = make_step(remat, torch.compile(remat, dynamic=False,
                                      mode='reduce-overhead'), bf16=True)
torch.cuda.reset_peak_memory_stats()
start = torch.cuda.memory_allocated()
speed = record('+ checkpointing',
               tokens_per_second(step, windows(corpus, 512)))
memory = (torch.cuda.max_memory_allocated() - start) / 2**30
print(f'memory beyond weights and optimizer state: {memory:.1f} GiB')
```

```
+ checkpointing: 556,486 tokens/s (0.85x)
memory beyond weights and optimizer state: 3.2 GiB
```

Checkpointing lowers throughput by a tenth to a quarter and cuts the memory of the step to less than half. It is the right change for a model or a minibatch that would not fit otherwise. For this step it is the wrong one, as the memory of the previous configuration predicted, and the configurations that follow leave it out.

13.7.3 Data Parallelism

The fastest single-GPU step still leaves three of the machine's four GPUs idle. We predict what data parallelism gains on them before we measure it.

13.7.3.1 Predicted Speedup

Four GPUs could at best process four times as many tokens per second as one. In data parallelism (Section 13.5), every GPU holds a copy of the model and runs the step on a minibatch of its own; before the update, the GPUs sum their gradients with an allreduce. The cost model of Section 13.5.4 puts the time of a step on k GPUs at $t_{\text{compute}}(B/k) + m\alpha + 2N/\beta$: the computation on each GPU's share of a minibatch of B examples, plus m allreduce calls with a fixed cost α each, which move the N bytes of the gradients at a bandwidth β.

Here the minibatch per GPU stays fixed as GPUs are added, which is called weak scaling, so the computation term stays the time of a step on one GPU. For the communication terms we substitute the measured time of one allreduce of a buffer as large as all gradients, and obtain

$$t_{\text{step}}(k) \approx t_{\text{step}}(1) + t_{\text{allreduce}}(k). \tag{13.13}$$

A framework that sums the gradients in several buckets makes a few calls instead of one, which adds a few fixed costs α; next to an allreduce of tens of milliseconds they are small. Since k GPUs process k minibatches in the time of Equation 13.13, throughput grows by the factor $k\,t_{\text{step}}(1)/(t_{\text{step}}(1) + t_{\text{allreduce}}(k))$. A second GPU therefore raises throughput only if the allreduce takes less time than a step.

The gradients of our GPT occupy 76 MB in fp32. Our GPUs have no direct peer-to-peer path, so the collective library moves every byte through host memory (Section 13.6), and an allreduce of 76 MB takes tens of milliseconds. That is as long as a compiled fp32 step at batch 64 or longer, but only a fraction of a bf16 step at batch 512. The cost model therefore predicts no gain from a second GPU for the first configuration and a substantial one for the second.

The model leaves out one effect. A framework can start summing the gradients of the last layers while the backward pass still computes those of the first (Figure 13.18). The prediction is therefore a floor, and the distance of a measurement above it shows how much of the allreduce was hidden behind computation.

13.7.3.2 Measured Speedup

We now test the prediction on two and four GPUs. `data_parallel(k, bf16, batch_size)` trains on k GPUs with the given precision and minibatch size per GPU. It returns the throughput, together with the time of one allreduce of a buffer as large as all gradients on the same GPUs.

Both configurations run the step without its host overhead, one in fp32 at batch 64 and one in bf16 at batch 512.

`DistributedDataParallel` runs one process per GPU (Section 13.6). The step therefore goes into a script, which `d2l.torchrun` of Section 13.6 starts on k GPUs. The script joins the process group with `d2l.init_distributed`, draws its minibatches with `windows` and builds the compiled step with `make_step`, both saved from this section, and wraps the model in `DistributedDataParallel`, which sums the gradients in buckets while the backward pass runs. It compiles without CUDA graphs: both configurations keep the device busy long enough to hide the work of the host. Its first process prints the throughput and the time of one allreduce. Every process draws its own windows. We run the script on one GPU as well, so that the single-GPU step is measured under the same conditions. The script's first process runs on GPU 0, where the CUDA graphs of the notebook's own compiled steps still hold memory, so the notebook releases that memory before the first launch.

```
DDP_GPT = r'''
import json, sys, torch
from torch.nn.parallel import DistributedDataParallel as DDP
from d2l import torch as d2l

rank, k, device = d2l.init_distributed()
bf16, batch_size = sys.argv[1] == 'bf16', int(sys.argv[2])
torch.set_float32_matmul_precision('high')
torch._inductor.config.compile_threads = 1  # k processes compile at once
data = d2l.WikiText103(batch_size, 128)
corpus = torch.tensor(data.train, device=device)
model = DDP(d2l.GPT(len(data.vocab), num_hiddens=512).to(device))
step = d2l.make_step(model, torch.compile(model, dynamic=False), bf16)
batches = d2l.windows(corpus, batch_size)
seconds = d2l.Benchmark(lambda: step(*next(batches)), 20, 50).time
grads = torch.zeros(d2l.num_params(model), device=device)
allreduce = d2l.Benchmark(lambda: torch.distributed.all_reduce(grads),
                          repeats=10).time
if rank == 0:
    print(json.dumps({'speed': k * batch_size * 128 / seconds,
                      'allreduce': allreduce}))
torch.distributed.destroy_process_group()
'''

# The CUDA graphs of the compiled steps above hold memory on GPU 0, which
# the process of rank 0 needs: reset the compiler and return the memory
torch.compiler.reset()
torch.cuda.empty_cache()

def data_parallel(k, bf16, batch_size):
    """Tokens per second on k GPUs, and seconds per allreduce of all
    gradients."""
    r = d2l.torchrun(DDP_GPT, k, 'bf16' if bf16 else 'fp32', batch_size)[0]
    return r['speed'], r['allreduce']
```

```
for bf16, batch_size in ((False, 64), (True, 512)):
    tokens = batch_size * 128  # Per GPU and step
    one, _ = data_parallel(1, bf16, batch_size)
    print(f'{"bf16" if bf16 else "fp32"}, batch {batch_size} per GPU: '
          f'{one:,.0f} tokens/s on one GPU')
    for k in (2, 4):
        if k <= d2l.num_gpus():
            speed, allreduce = data_parallel(k, bf16, batch_size)
            predicted = k * tokens / (tokens / one + allreduce)
            print(f'  {k} GPUs: allreduce {1e3 * allreduce:3.0f} ms, '
                  f'speedup predicted {predicted / one:.2f}x, '
                  f'measured {speed / one:.2f}x')
```

```
fp32, batch 64 per GPU: 365,164 tokens/s on one GPU
  2 GPUs: allreduce  31 ms, speedup predicted 0.84x, measured 0.91x
  4 GPUs: allreduce  48 ms, speedup predicted 1.27x, measured 1.36x
bf16, batch 512 per GPU: 663,091 tokens/s on one GPU
  2 GPUs: allreduce  31 ms, speedup predicted 1.52x, measured 1.63x
  4 GPUs: allreduce  48 ms, speedup predicted 2.69x, measured 2.93x
```

At batch 64 the allreduce takes about as long as the step or longer, so two GPUs are predicted to be no faster than one. At batch 512 in bf16 the step is several times longer, while the allreduce does not depend on the minibatch. The prediction rises to about three quarters of linear scaling on two GPUs, and somewhat less on four, where the allreduce moves more data through each GPU. No measurement lies more than a few percent below its prediction, and most lie above it, by the part of the allreduce that ran during the backward pass, which brings them closer to linear scaling. On this machine, data parallelism therefore helps only once the step on each GPU is long.

GPUs connected by NVLink move data more than ten times faster than our path through host memory (Table 13.2); the allreduce then shrinks by the same factor, and the cost model predicts most of the linear speedup even at batch 64.

13.7.4 Results

We now put the measurements of all the changes side by side and check that the fastest step still trains the model.

13.7.4.1 Cumulative Comparison

The table collects the single-GPU configurations in the order in which we applied them. Each ratio in the column `step` belongs to one change, measured on top of all the changes above it; the column `total` is the product of the ratios so far. The last column converts throughput into arithmetic with the count of six FLOPs per parameter and token.

```
flops = 6 * num_params  # Per token, without the attention scores
print(f'{"configuration":16}{"tokens/s":>11}{"step":>7}{"total":>7}'
      f'{"TFLOP/s":>9}')
first = previous = next(iter(results.values()))
for name, speed in results.items():
    print(f'{name:16}{speed:11,.0f}{speed / previous:6.2f}x'
```

```
            f'{speed / first:6.2f}x{speed * flops / 1e12:9.1f}')
    previous = speed
```

```
configuration      tokens/s   step  total  TFLOP/s
baseline            327,847  1.00x  1.00x     37.4
+ compile           368,795  1.12x  1.12x     42.1
+ bf16              644,090  1.75x  1.96x     73.5
+ batch 512         653,408  1.01x  1.99x     74.5
+ checkpointing     556,486  0.85x  1.70x     63.5
```

The three changes before checkpointing together raise throughput about twofold or more. At batch 512 the step performs roughly half of the 165 TFLOP/s that the tensor cores can execute in bf16. Checkpointing then lowers the throughput again.

Each ratio depends on its place in the sequence. A change that shortens the work of the device gains nothing while the step waits for the host, and a change that removes host work gains nothing while the device is busy. A ranking of techniques that measures each one alone on the same baseline can therefore mislead.

13.7.4.2 Training Check

Throughput says nothing about whether the faster step still trains the model. We therefore train a fresh model with the final configuration, bf16 at batch 512, for 300 steps or about 20 million tokens, and then compute its loss on the validation text with `d2l.lm_perplexity` (Section 12.1), in nats per byte. Two numbers from Section 11.2 set the scale. A bigram model, which predicts each byte from the previous one, reaches 2.45 nats per byte; the GPT of width 256 reached about 1.35 after 1,000 steps of 8,192 tokens with a tuned learning-rate schedule.

```
learner = d2l.GPT(len(data.vocab), num_hiddens=512).to(device)
step = make_step(learner, torch.compile(learner, dynamic=False,
                                        mode='reduce-overhead'), bf16=True)
batches = windows(corpus, 512)
losses = torch.stack([step(*next(batches)).detach()
                      for _ in range(300)]).tolist()
validation = math.log(d2l.lm_perplexity(learner, data.val_dataloader()))
print(f'training loss {losses[0]:.2f} at the first step, '
      f'{sum(losses[-20:]) / 20:.2f} over the last 20 steps; '
      f'validation loss {validation:.2f}')
assert validation < 2.45  # Better than the bigram model
```

```
training loss 5.61 at the first step, 1.82 over the last 20 steps; validation loss 1.80
```

The training loss starts near $\log 256 \approx 5.55$, the loss of a uniform guess over the 256 bytes, and the validation loss ends well below the bigram level. The faster step trains the model.

The fastest configuration still leaves about half of the card's bf16 peak unused. What remains lies in the memory-bound kernels between the matrix products, which the third profile showed, in the efficiency of the products themselves, and in the width of the numbers. The tensor cores of the RTX 4090 run fp8 at twice the bf16 rate, but training in fp8 needs a scaling factor for each tensor or block of values to keep them within its narrow range. That recipe, kernels written for a particular model, and training on more GPUs than one machine

holds belong to the Language Models part. The method stays the same there: measure, find the limit, change one thing, check, and measure again.

13.7.5 Summary

A training step can be made about twice as fast or more by a sequence of changes, each chosen by a measurement. A profile classifies the step: a device that waits for the host calls for less work on the host per step, a device busy with many short elementwise kernels calls for fusion, and a busy device dominated by matrix products calls for faster arithmetic. On one RTX 4090, fusion or less host work, bf16, and a larger minibatch together raised the throughput of our 19-million-parameter GPT about twofold or more. Activation checkpointing lowered it, because memory did not limit the step. Every change to the computation is checked for correctness before it is timed, every change is measured on top of the previous ones, and a short training run confirms that the fastest step still trains the model. The cost model $t_{\text{step}}(1) + t_{\text{allreduce}}(k)$ predicts, before any measurement, that on GPUs connected through host memory data parallelism gains little for a short step and most of the linear speedup for a long one.

13.7.6 Exercises

1. code **Noise floor.** Each throughput in Section 13.7.4 comes from a single window of 50 steps.
    1. Call `tokens_per_second` ten times in a row for the second configuration of the table, and report the mean and the relative standard deviation of the ten results.
    2. Estimate the ratio of the third configuration to the second twice: from two blocks of ten consecutive windows, and from ten pairs of alternating windows. Suppose that the GPU clock drifts slowly during the measurement. Explain which of the two designs cancels the drift, and compare the two estimates.
    3. Suppose that windows vary independently with the relative standard deviation of the first part. Find the number of windows per configuration that resolves a change of 3% at two standard errors of the ratio. Which ratios in the column `step` does one window per configuration resolve?
2. code **Model FLOPs utilization.** The GPT of Section 13.7.1 has $L = 6$ blocks of width $d = 512$ and reads contexts of $T = 128$ tokens. Up to the rounding of the SwiGLU width, each block holds $12d^2$ weights in its matrix products, and training costs about 6 FLOPs per weight and token, which is the count of the table in Section 13.7.4.
    1. The scores $\mathbf{Q}\mathbf{K}^\top$ and the weighted sum of the values are matrix products without weights. Count their training FLOPs per token and block, taking the backward pass as twice the forward pass and counting the masked scores. Show that their ratio to the weight FLOPs is $T/(6d)$, and evaluate the ratio for this GPT and for a context of 1,024 tokens.
    2. Recompute the last column of the table with these FLOPs included, and the fraction of the peak of each configuration's format that it reaches: 165 TFLOP/s in bf16 and 83 TFLOP/s in tf32.
    3. A utilization figure can count the FLOPs that the model requires or the FLOPs that the hardware executes. Name two configurations or kernels of Section 13.7.2 for which the two counts differ, and state the direction of each difference.

3. code **Fixed cost of a step.** Model the time of the bf16 step at a minibatch of B windows as $t(B) = a + bB$, a fixed cost a plus a cost b per window.
    1. Measure the step at $B = 32, 64, 128, 256, 512$, fit a and b by least squares, and report the fraction of the step at $B = 64$ that a accounts for.
    2. The AdamW update reads every weight, its gradient, and both moment estimates, and writes the weight and both moments, all in fp32. Compute the bytes that it moves per step for this GPT and the time they take at the 1.0 TB/s of the RTX 4090. What fraction of a is this?
    3. Use the fit to predict the throughput at $B = 1{,}024$ and $B = 2{,}048$. Check with the memory measured in Section 13.7.2 whether these minibatches fit on the card, and measure those that do.
4. code **Causality gate.** The training run at the end of Section 13.7.4 accepts the final configuration when its validation loss lies below that of a bigram model. Consider an optimized attention kernel that silently ignores the causal mask.
    1. Predict how the training loss of the GPT behaves with this kernel and whether the check passes. Explain why the loss on the validation windows does not detect the defect either.
    2. Write a gate that changes the last input token of every sequence in a minibatch and compares the logits at all earlier positions before and after the change. Run it on `learner` and on a copy of the model whose attention omits the mask, and justify the tolerance that you choose.
5. **Throughput versus time to target.** The cumulative comparison ranks configurations by tokens per second, but what a training run needs is a short time to reach a target loss. The model of Section 9.8 says that training at batch size b needs

$$E(b) = E_{\text{min}} \left(1 + \frac{b}{b_{\text{noise}}}\right)$$

examples to reach the target, where E_{min} is the number needed with very small batches and b_{noise} is the noise scale of the gradient.

    1. Let $\rho > 1$ be the ratio of the throughput at batch 512 to the throughput at batch 64. Show that batch 512 reaches the target sooner exactly when

$$b_{\text{noise}} > \frac{512 - 64\rho}{\rho - 1}, \tag{13.14}$$

    and evaluate the threshold for $\rho = 1.2$ and for $\rho = 1.7$.
    2. Classify each configuration of Section 13.7.2, data parallelism under weak scaling, and a switch to the optimizer of Section 9.7 by whether it changes $E(b)$, the throughput, or both.
6. code **Width 256.** At `num_hiddens=256` the GPT is the model that Section 11.2 trains.
    1. Consider the product of the 8192×256 activations of one minibatch with a 256×256 weight matrix, the shape of the output projection $\mathbf{W}_o$. Compute its arithmetic intensity with 4-byte tf32 operands and with 2-byte bf16 operands, place both on the roofline of Figure 13.1 for the RTX 4090, taking the tf32 peak as half the bf16 peak, and predict the speedup of bf16 for this product.

2. Run the baseline and the first two changes of Section 13.7.2 at `num_hiddens=256`. Profile the second and the third configuration with `profile`, and split the device time of each between the matrix products and all other kernels. Account for the measured ratio of the two configurations kernel by kernel, and compare it with the prediction.

7. code **Data-parallel overlap.** ▯ Let ϕ be the fraction of $t_{\text{allreduce}}(k)$ that the framework hides under the backward pass, so that a step on k GPUs takes

$$t_{\text{step}}(k) = t_{\text{step}}(1) + (1-\phi)\, t_{\text{allreduce}}(k). \tag{13.15}$$

   1. Compute ϕ at $k = 2$ and $k = 4$ for both configurations of Section 13.7.3 from the printed measurements. What does a negative value of ϕ indicate?
   2. Repeat the bf16 measurement with 128 and 256 windows per GPU, and plot the weak-scaling efficiency and ϕ against the minibatch per GPU. Explain the trend by the terms of Equation 13.13.
   3. Predict from Equation 13.13, without overlap, the smallest minibatch per GPU at which four GPUs reach 90% of linear scaling in bf16, and check the prediction.

8. [extended] **Another model.** ▯▯ Apply the method of this section to the Mamba language model of Section 12.3.3.4 or to `d2l.ViT` of Section 11.5.

   1. Write the rules of the measurement in the form of the table in Section 13.7.1, profile the baseline, and classify its limiting regime in the terms of Section 13.1.3 before choosing the first change.
   2. Apply the removal of host overhead, bf16, a larger minibatch, and activation checkpointing one at a time, with a correctness check before every timing, and tabulate the cumulative comparison.
   3. Identify every change whose effect differs in sign or size from the GPT, and explain each difference by the mechanism that the change targets.

13.8 Further Reading

The sections of this chapter teach one representative of each idea: one roofline with one memory bandwidth, one GPU and its links, one compiler per framework, three ways to save memory, one collective algorithm, and one transformer made fast on one machine. This section collects, in the order of the chapter, topics they had no room for. They range from finer performance models and other accelerators to hand-written kernels, further ways to save memory, other collective algorithms, older forms of distributed training, and the topics beyond one machine that the Language Models part takes over. Each topic states the problem it addresses and what it changes relative to the chapter. Implementations, tutorials, and courses for the material that the chapter does teach are listed in its introduction (Chapter 13). Nothing else in the book depends on this section.

13.8.1 Measuring and Modeling

13.8.1.1 One Roof per Memory Level

The roofline of Section 13.1 has a single bandwidth β, that of GPU memory. Data that a kernel reuses from the L2 cache or from shared memory arrives faster, so a kernel with a low intensity against GPU memory can still run above the roof that this intensity predicts. The

cache-aware roofline (Ilic et al. 2014) draws one bandwidth slope per level of the memory hierarchy and counts the bytes each level delivers. Yang et al. (2020) collect the FLOPs and the bytes per level from hardware counters with Nsight Compute and place the kernels of a deep learning model against an L1, an L2, and a memory roof at once. The single number I of the chapter becomes one intensity per level, and the level whose roof is lowest for its intensity limits the kernel.

13.8.1.2 Latency Hiding

The bandwidth of Section 13.2.1 is reached only when enough requests are in flight to cover the latency of each one, as the exercise on Little's law in Section 13.2 computes. A GPU keeps them in flight by switching among many resident warps. Volkov (2016) shows that the number of warps needed follows from the latency and the throughput of each unit, that independent instructions within one thread can replace additional warps, and that common GPU performance models mispredict throughput because they ignore this. The analysis explains why small kernels and kernels with few threads stay below the bandwidth roof that their intensity predicts.

13.8.1.3 Device Timers and System Profilers

The `Benchmark` of Section 13.1 waits for the device and reads the host's clock, so it measures what a step costs in wall-clock time. CUDA events (`torch.cuda.Event(enable_timing=True)`) take timestamps on the GPU's own stream instead. They time a region of device work without draining the queue, which suits measurements inside a training loop, but they miss host time and so cannot detect the overhead-bound regime. The framework profilers of Section 13.1.4 attribute device time to kernels. NVIDIA's Nsight Systems records the whole timeline of a process, with Python threads, CUDA calls, kernels, copies, and NCCL collectives on one axis, and shows idle gaps whose cause lies outside the framework. Nsight Compute measures a single kernel's achieved FLOP/s and bytes against both roofs.

13.8.1.4 The Time of a Transformer Step

The profile of Section 13.7 found the matrix products to be the largest share of a step but not all of it. Ivanov et al. (2021) made the same measurement on BERT. The tensor contractions perform over 99% of the FLOPs of a training step but take only 61% of its runtime, and over a third of the runtime goes to normalizations and elementwise operations, which are bandwidth-bound. Fusing these operators and choosing data layouts that suit each kernel reduced the memory traffic and made a BERT encoder layer 1.3 times faster than the frameworks of the time. The study applies the bandwidth regime of Section 13.1 to a whole model and makes the case for the compilers of Section 13.3.

13.8.1.5 Utilization as a Reported Number

The case study ends with the fraction of the peak that its fastest step reaches. Reports of large training runs quote the same quantity as *model FLOPs utilization* (MFU) (Chowdhery et al. 2022). MFU is the FLOPs that the model needs per token, times the tokens per second, divided by the peak of the hardware. *Hardware FLOPs utilization* also counts the FLOPs of recomputation, so activation checkpointing (Section 13.4.3) raises it while it lowers MFU (Korthikanti et al. 2022). Only MFU compares systems fairly, because it credits only the work that the model needs.

13.8.1.6 Time to a Target Loss

Throughput in tokens per second rewards any change that makes a step faster, even one that makes training need more steps. The modded-nanoGPT speedrun (Jordan and contributors 2024) ranks changes by the wall-clock time to train a small GPT to a fixed validation loss on one node of eight GPUs. Its records combine the tools of this chapter (compiled and fused kernels, fused attention, fp8 products for some layers) with changes to the optimization, such as the Muon optimizer of Section 9.7. Under this measure a larger minibatch can lose even when it raises throughput, once it grows past the critical batch size of Section 9.8.

13.8.2 Hardware and Number Formats

13.8.2.1 Other Accelerators

The table of Table 13.2 lists NVIDIA parts only. Accelerators from other vendors follow the same design of stacked memory beside a die of matrix units, so the roofline describes them with two different numbers. In late 2026, AMD's Instinct MI355X has 288 GB of HBM3e at 8 TB/s and runs 2,517 dense bf16 TFLOP/s, a ridge point of about 310 FLOP/byte. Google's TPU7x (Ironwood) has 192 GiB at 7.4 TB/s and 2,307 bf16 TFLOP/s, a ridge point of about the same size. Intel's Gaudi 3 has 128 GB at 3.7 TB/s and runs 1,678 TFLOP/s in bf16 and in fp8 alike. AWS's Trainium2 has 96 GiB at 2.9 TB/s and 667 bf16 TFLOP/s. The vendors' datasheets give these values.

The ridge points of these accelerators range from about 230 FLOP/byte (Trainium2) to about 450 (Gaudi 3), against 281 and 295 for the B200 and the H100. The rates per format differ more: fp8 runs no faster than bf16 on Gaudi 3 and on TPU v6e, and fp6 runs at the fp4 rate on the MI355X but at the fp8 rate on NVIDIA's Blackwell GPUs.

13.8.2.2 Unified Memory

A single stream of text is decoded at a rate bounded by the memory bandwidth divided by the bytes of the weights (Equation 13.4), and only if the weights fit at all. Apple's M3 Ultra shares up to 512 GB of memory between its CPU and GPU at 819 GB/s. A model of 70 billion parameters stored in 8 bits occupies about 70 GB: it does not fit on an RTX 4090, while on the M3 Ultra the bound allows about a dozen tokens per second. Machines of this kind trade a low compute roof for capacity, which suits decoding a single stream of a large model and suits training poorly. The Tools appendix (Section 39.4) covers the choice between such machines and GPUs.

13.8.2.3 Block-Scaled Formats in Detail

Section 13.2 explains why 4-bit formats need a scale for every small block of values. The OCP microscaling formats (Rouhani et al. 2023) share one power-of-two scale, stored in 8 bits, among 32 values; the family covers 8-, 6-, and 4-bit floating point and 8-bit integers, and adds a quarter bit per value. NVIDIA's NVFP4 shares a scale in fp8 among 16 values and adds one fp32 scale per tensor. It spends half a bit per value on scales, and its scales need not be powers of two, so they follow outliers more closely. Which format runs at which rate is a decision of each chip's designers, as the accelerators above show.

13.8.3 Compilers and Kernels

13.8.3.1 Writing Fused Kernels

The compilers of Section 13.3 fuse chains of elementwise operations and reductions, but they do not find algorithmic rewrites such as FlashAttention. Triton (Tillet et al. 2019) makes a block of a tensor the unit of programming. The author of a kernel decides the tiling and which values stay on the chip, and the compiler handles the threads, the vector instructions, and the layout in shared memory. Inductor emits Triton for the GPU kernels it generates, so its output is a readable starting point. Pallas plays the same role in JAX, lowering through Triton or Mosaic GPU on GPUs and through Mosaic on TPUs. The successive versions of FlashAttention show how a kernel follows the hardware. FlashAttention-2 (Dao 2024) divides the work among thread blocks by query tiles, and FlashAttention-3 (Shah et al. 2024) overlaps data movement with the matrix products and the exponentials on Hopper GPUs and adds fp8.

13.8.3.2 Compile Time at Scale

The break-even point of Section 13.3.3 counts the time a compiler takes. For a large model this time grows to minutes, and it recurs in every process and for every new shape. Regional compilation compiles one repeated block, such as a transformer layer (`torch.compiler.nested_compile_region`), and reuses the result for every copy. Compilation caches keep compiled programs across processes: `torch.compiler.save_cache_artifacts` and `load_cache_artifacts` in PyTorch, and the persistent compilation cache (`jax_compilation_cache_dir`) in JAX. Autotuning goes the other way and spends more compile time for faster kernels. `torch.compile(mode='max-autotune')` times several tilings and kernel templates for each matrix product and keeps the fastest, and XLA autotunes its GPU matrix products and fusions in the same way.

13.8.3.3 Kernel Replay beyond Inference

Section 13.3.4 records and replays a forward pass. A training step adds the backward pass, the optimizer update, and buffers that persist from step to step. The CUDA graph trees behind `mode='reduce-overhead'` record the forward and backward passes as separate graphs that share one memory pool, record again for every new input shape, and overwrite the outputs of the previous replay, so a training loop marks the start of each step (`torch.compiler.cudagraph_mark_step_begin()`). XLA records runs of kernels as CUDA graphs, which it calls command buffers, by default.

13.8.3.4 Exporting a Captured Graph

Section 13.3 captures a graph only to compile it for the same Python process. The same graph can be saved and run without Python. `torch.export` captures a whole program with no graph breaks, failing where `torch.compile` would break, and replaces the older TorchScript. A jitted JAX function lowers to StableHLO, a versioned format that other compilers and runtimes accept. Exported graphs are how models reach phones, browsers, and serving systems, which the Language Models part takes up.

13.8.3.5 Earlier Graph-Building Designs

Theano (Bergstra et al. 2010) and TensorFlow 1 (Abadi et al. 2016) made the user build a symbolic graph and then run it, which gave the compiler the whole program but made the

program hard to write and to debug. Eager frameworks reversed the choice, and the tracing compilers of Section 13.3 recover the graph from eager code. Frostig et al. (2018) describe the tracing of Python functions into XLA from which `jax.jit` grew.

13.8.4 Memory

13.8.4.1 Smaller Optimizer State

Of the 16 bytes per parameter of Section 13.4.1, Adam's two moment estimates take 8. Eight-bit Adam (Dettmers et al. 2022) stores both in 8 bits, quantized block by block so that one large entry affects only its own block, and matches the results of the fp32 optimizer with 2 bytes per parameter instead of 8. Adafactor (Shazeer and Stern 2018) keeps, for each weight matrix, only the sums of the second moment over its rows and over its columns, and reconstructs the moment of each entry from the two. Its state grows with the sum of a matrix's dimensions instead of their product.

13.8.4.2 Offloading to Host Memory and Disk

Host memory is larger than GPU memory by an order of magnitude and cheaper per byte. ZeRO-Offload (Ren et al. 2021) keeps the optimizer state in host memory and runs the update on the CPU, so the GPU holds the weights and the activations only. The drawback is traffic over PCIe. In fp32 the gradients travel to the host and the updated weights return, 8 GB per step for a model of a billion parameters, which takes a sizable fraction of a second at the 23 GB/s of Section 13.2.4.1. Large minibatches hide this time behind the computation. ZeRO-Infinity (Rajbhandari et al. 2021) extends the hierarchy to NVMe drives. JAX exposes offloading through its checkpoint policies, which can send named activations to host memory instead of discarding them.

13.8.4.3 Choosing What to Recompute

Section 13.4.3 stores the input of every block and recomputes the rest. Which values to store under a memory budget is an optimization problem. For a chain of identical steps, binomial checkpointing (Griewank and Walther 2008) minimizes the recomputation for a given number of stored states. Checkmate (Jain et al. 2020) solves the problem for a general graph as an integer linear program, with the measured cost of each operation. Compilers make the choice themselves: when AOTAutograd splits the joint graph of the forward and backward pass, it recomputes cheap elementwise results in the backward pass instead of storing them, and its memory budget (`torch._functorch.config.activation_memory_budget`) sets how far it goes.

13.8.4.4 Fragmentation

The memory that a training step needs and the memory that it can obtain differ by fragmentation. PyTorch's caching allocator keeps freed blocks for reuse, and when the shapes change from step to step, as with variable sequence lengths, free memory splits into pieces that no new tensor fits. An out-of-memory error then reports a large amount "reserved but unallocated". Setting `PYTORCH_CUDA_ALLOC_CONF=expandable_segments:True` lets the allocator grow a segment in place instead of allocating new ones, which removes most of this waste. JAX allocates most of the device's memory on its first use of a GPU (Section 5.7) and manages it in its own pool. XLA plans the buffers of a compiled program ahead of time, which leaves the allocator fewer and larger requests.

13.8.5 Communication

13.8.5.1 Collective Algorithms beyond the Ring

The ring allreduce of Section 13.5.3 moves the fewest bytes, but it takes $2(k-1)$ sequential steps, each with a fixed cost. For small messages on many devices that cost dominates. Trees reduce and broadcast in about $\log_2 k$ steps, and NCCL combines two binary trees in which every device is a leaf in one and an inner node in the other, which keeps all links busy. Recursive halving and doubling, a reduce-scatter and an all-gather in $\log_2 k$ steps each, reach the bandwidth of the ring with the step count of a tree when k is a power of two (Thakur et al. 2005). Collective libraries pick among these algorithms by message size and topology, and machines with several levels of links run a ring at each level.

13.8.5.2 Bus Bandwidth

Collective benchmarks such as NVIDIA's nccl-tests report two rates. The *algorithm bandwidth* is the message size divided by the time. The *bus bandwidth* of an allreduce is the algorithm bandwidth times $2(k-1)/k$, the rate at which each link of an ideal ring carries data, so it can be compared across k. The β of Section 13.5 is defined by the time $2N/\beta$ of a large allreduce of N bytes, so it is twice the algorithm bandwidth. It equals the bus bandwidth times $k/(k-1)$, twice the bus bandwidth at $k = 2$.

13.8.5.3 Parameter Servers and Asynchronous Updates

Before collectives became the default for dense models, large-scale learning summed gradients on parameter servers: machines that each hold a shard of the parameters and answer two requests, *push* a gradient into the sum and *pull* the current values (Li, Andersen, et al. 2014). Sharding the parameters over many servers lets the bandwidth of the aggregation grow with the cluster, and the interface admits updates computed from stale parameters. DistBelief (Dean et al. 2012) trained with such asynchronous updates. Asynchrony and the related local SGD, in which each replica takes several steps between averages, communicate less often instead of faster, and both change the optimization as well as its speed. Synchronous data parallelism therefore remains the default for dense models. Parameter servers survive where the parameters are too large and too sparse for every device to hold them, as in the embedding tables of recommender systems (Chapter 42).

13.8.5.4 Large Minibatches

Weak scaling (Section 13.5) keeps the minibatch per GPU fixed, so the global minibatch grows with the number of GPUs, and the learning rate has to follow. Goyal et al. (2017) trained a ResNet-50 on ImageNet with minibatches of 8,192 images to the accuracy of small minibatches by scaling the learning rate in proportion to the minibatch and warming it up over the first epochs. Section 9.8 explains why the gain stops at the critical batch size.

13.8.5.5 Parallelism Interfaces

PyTorch offers each form of parallelism as its own interface: `DistributedDataParallel`; `fully_shard` for sharded data parallelism; `torch.distributed.tensor.parallel` for splitting the matrices of a layer across devices; and `torch.distributed.pipelining` for splitting the layers into stages. A device mesh with several dimensions combines them, for example sharding within a machine and replicating across machines. JAX expresses all of them as placements

of arrays on a mesh with named axes, and the compiler's partitioner inserts the collectives, a design that GSPMD (Xu et al. 2021) introduced.

13.8.6 Beyond One Machine

The Language Models part takes up the topics that need several machines or serve trained models; each gets one line here. Tensor, pipeline, and expert parallelism split a model too large for data parallelism (Shoeybi et al. 2019; Narayanan et al. 2021; Lepikhin et al. 2021). Training in fp8 and fp4 needs scales per tensor or per block and fp32 accumulation (Micikevicius et al. 2022; DeepSeek-AI, Liu, et al. 2024). Serving engines batch requests continuously and page the key–value cache (Kwon et al. 2023). Quantization compresses trained weights to 4 or 8 bits for inference.

Chapter 14

Reinforcement Learning

Every model in this book so far was trained on a dataset that sat still: nothing the model predicted changed which example came next. Now consider a language model working through a mathematics problem. It writes a thousand tokens of reasoning, and only at the end does a checker say whether the answer is right. No token has a label of its own, and each one changes what the model can write next. A controller balancing a pole on a moving cart is in the same position: each push changes the angle it must correct next, and the only reward is the time the pole stays up. Figure 14.1 draws the loop between the decision maker, the *agent*, and the world it acts on, the *environment*.

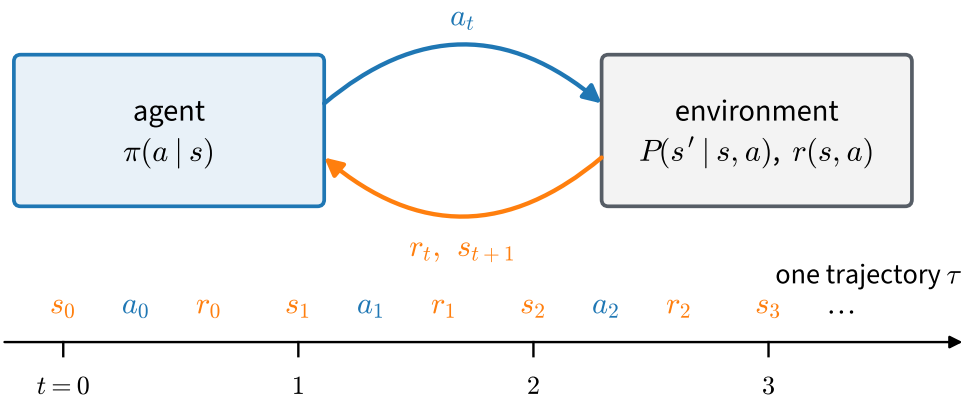


Figure 14.1: The loop between an agent and its environment. At step t the agent, following its policy π, chooses an action a_t (blue); the environment, governed by its transition probabilities P and reward r, returns a reward r_t and the next state s_{t+1} (orange). Repeating the exchange produces the trajectory $\tau = (s_0, a_0, r_0, s_1, a_1, r_1, \ldots)$.

Two questions follow. Which decisions deserve credit for a reward that arrives late? And how can a learner find good decisions when nobody shows it what they are? Richard Bellman answered the first in the 1950s for problems whose rules are known: score each situation by the reward still to come, and a long problem becomes a chain of one-step problems (Bellman 1957b). In the 1980s Andrew Barto, Richard Sutton and Chris Watkins learned the

same scores by trial and error, with the pole on a cart as an early test (Barto et al. 1983; Sutton 1988; Watkins 1989). Ronald Williams showed how to improve the rule that picks decisions, the *policy*, directly from the rewards its attempts earn (Williams 1992). Because the learner collects its own data, its mistakes change what it learns from next.

These ideas train controllers, game players and, after pretraining, language models that reason toward answers a checker accepts. They also explain why imitation is not enough: a model fine-tuned on an expert's text never practices recovering from its own mistakes.

Section 14.1 asks how to write such a problem down. Section 14.2 asks how to find the best policy when the rules are known. Section 14.3 asks what can be learned from seeing only where each action leads. Section 14.4 asks how often to try actions that look worse. Section 14.5 asks what an expert's demonstrations teach, and why copying them fails over a long task. Section 14.6 asks how a policy can improve from its own attempts. Section 14.7 asks how to make that estimate less noisy, the question behind the group-based updates that train language models. Section 14.8 asks what changes when a neural network replaces the table, and it ends with a map of the methods of this chapter and the next.

The code is in PyTorch and JAX, and most examples are small enough to solve exactly, so that every estimate can be checked against the exact answer. Chapter 15 continues with value networks inside the update, reuse of past experience and fixed datasets. Problems with several agents are left out, and hidden state and learned models of the environment appear only briefly. The Language Models part treats post-training, including learning from human preferences, in more depth.

Resources and Further Reading

The following freely available resources provide theoretical treatments, courses, implementations, and empirical guidance.

Textbooks

- Sutton and Barto, *Reinforcement Learning: An Introduction* (Sutton and Barto 2018) provides a comprehensive introduction to value methods, policy methods, planning, and function approximation. Sutton and Barto received the 2024 Turing Award for their foundational contributions to reinforcement learning.
- Szepesvári, *Algorithms for Reinforcement Learning* (Szepesvári 2010) states this chapter's algorithms and their guarantees in under a hundred pages.
- Bertsekas, *A Course in Reinforcement Learning* (Bertsekas 2025) develops reinforcement learning from the perspective of optimal control and dynamic programming.
- Agarwal, Jiang, Kakade, and Sun, *Reinforcement Learning: Theory and Algorithms* (Agarwal et al. 2019) supplies the sample-complexity rates this chapter states qualitatively.
- Lattimore and Szepesvári, *Bandit Algorithms* (Lattimore and Szepesvári 2020) is the full theory behind Section 14.4.

Courses

- Berkeley CS285: Deep Reinforcement Learning covers policy gradients, value methods, model-based learning, and offline reinforcement learning at the graduate level.
- Stanford CS234: Reinforcement Learning is the course closest in scope to this chapter: foundations, exploration, and policy gradients with their guarantees.
- Stanford CS224R: Deep Reinforcement Learning covers the practice, imitation and offline methods included.

- CMU 10-703: Deep Reinforcement Learning and Control pairs the algorithms with control and physical embodiment.
- MIT 6.7920: Reinforcement Learning: Foundations and Methods develops the dynamic-programming theory introduced in Section 14.2.

Annotated implementations

- CleanRL (Huang, Rousslan F. J. Dossa, et al. 2022) provides single-file, benchmarked implementations of the major deep reinforcement learning algorithms discussed after Section 14.8.
- Stable-Baselines3 (Raffin et al. 2021) provides library implementations of the same algorithms, with tuned hyperparameters in its accompanying zoo.
- Gymnasium (Towers et al. 2024) documents the environment interface used throughout these chapters.

Empirical practice

- Spinning Up in Deep RL (Achiam 2018) is a short curriculum from the score function to SAC, written to be read beside running code.
- The 37 Implementation Details of Proximal Policy Optimization (Huang, Rousslan Fernand Julien Dossa, et al. 2022) documents how implementation choices omitted from algorithm descriptions affect PPO performance and reproducibility.

14.1 Markov Decision Processes

Picture a robot crossing a frozen lake, a program playing a board game, or a language model writing an answer one token at a time. Each choice changes what happens next: the robot slips into a different cell, the game reaches a different position, the answer takes a different turn. The reward may come only at the end, and the same command may have several outcomes. Before we can ask how to act well, we need a way to write such a problem down: what the decision maker knows at each step, what it can do, what follows, and what it is trying to achieve.

Bellman gave this description its form in a 1957 paper on Markovian decision processes (Bellman 1957a). His idea was to put everything about the past that matters for the future into a *state*, so that the consequences of an action depend only on the state in which it is taken. That one assumption makes the problem tractable. Once the future depends on the present state alone, a single number per state can say how good it is to be there. The resulting model, the *Markov decision process* (MDP), became a standard tool of operations research (Puterman 1994). Every method in this chapter and the next works on it, from dynamic programming to the training of language models with a reward.

We work on a sixteen-cell frozen lake, small enough that every quantity can be checked by hand. It still has the feature that makes acting over time hard: on slippery ice the agent often slides somewhere other than where it meant to go. We read the lake's transition table out of a simulator and name its parts, then define the return an agent maximizes and the discount that keeps it finite. We ask what a state must contain, and we examine a bonus that looks helpful but pays the agent for going in circles. A second, even smaller example closes the section: a language model answering arithmetic questions, scored by a checker.

The chapter uses the symbols of Table 14.1. Those of this section are defined below; the others belong to later sections, which define each where it is first used.

Table 14.1: Symbols of the reinforcement-learning chapters. The rows from V^π on belong to later sections, and those from λ on to Chapter 15. The exploration rate ϵ and the per-step disagreement ε are different symbols, and so are a response y and a value target y_t. A μ with a subscript or argument other than 0 is a mean (an arm's, a Gaussian's).

Symbol	Meaning	Introduced in
$\mathcal{S}$, $\mathcal{A}$	the sets of states and actions	Section 14.1
s_t, a_t, r_t	the state, action and reward at step t	Section 14.1
$P(s' \mid s, a)$	transition kernel: the probability of reaching s' from s under action a	Section 14.1
$r(s, a)$	expected reward of action a at state s	Section 14.1
μ_0	start-state distribution (for language models, the distribution of prompts)	Section 14.1
τ	trajectory $(s_0, a_0, r_0, s_1, \ldots)$	Section 14.1
γ	discount factor	Section 14.1
$R(\tau)$	return of a trajectory, $\sum_t \gamma^t r_t$	Section 14.1
T	number of steps of an episode, or a time limit	Section 14.1
$\pi(a \mid s)$, $\pi(s)$	policy: probability of action a at state s; the action of a deterministic policy	Section 14.1, Section 14.2
$\ell(s)$	Manhattan distance from cell s to the goal	Section 14.1
$\Phi(s)$	potential of a shaping bonus	Section 14.1
x, y, $r(x, y)$	prompt, response, and the verifier's score of the response	Section 14.1
y_t	target of a value update: an estimate of the return from step t	Section 14.1, Section 14.8
V^π, Q^π	expected return under π from a state, from a state-action pair	Section 14.2
V^*, Q^*	the same under an optimal policy	Section 14.2
$\mathcal{T}$	Bellman operator: one step of lookahead applied to a vector of values	Section 14.2
δ, δ_t	temporal-difference error of one transition, of the transition at step t	Section 14.3
α	step size of temporal-difference updates	Section 14.3

Symbol	Meaning	Introduced in
ϵ	exploration rate of ϵ-greedy action selection	Section 14.3
π_e	behavior policy, the one that chooses the actions while a method learns	Section 14.3
κ	weight of the uncertainty bonus	Section 14.4
K	number of responses sampled per prompt	Section 14.4
π_θ, θ	policy with parameters θ	Section 14.1 (the verifier's preference table), Section 14.5
d_t^π, d_γ^π	distribution of s_t under π; its discounted average over t	Section 14.5, Section 14.6
ε	per-step probability that a cloned policy disagrees with the expert	Section 14.5
$J(\theta)$	expected return of π_θ	Section 14.6
η	step size of a policy update	Section 14.6
$\hat{G}_t$	reward-to-go: the discounted sum of the rewards from step t on	Section 14.7
b	baseline subtracted from a return	Section 14.7
η_V	step size of a value-baseline update	Section 14.7
A^π	advantage $Q^\pi - V^\pi$	Section 14.7
λ	weight of the λ-return	Section 15.1
w, w^-	parameters of a value network; of its frozen or Polyak-averaged copy	Section 15.1, Section 15.4
n	depth of an n-step target, or of a plan	Section 15.1, Section 15.6
ρ_t	importance ratio $\pi_\theta(a_t \mid s_t)/\pi_{\theta_{\text{old}}}(a_t \mid s_t)$ at step t	Section 15.2
ϵ_{clip}	half-width of the clip band of the ratio	Section 15.2
Δ_{KL}	a KL budget per update	Section 15.2
π_{ref}, β	reference policy; weight of the KL penalty, and soft actor-critic's temperature	Section 15.3, Section 15.5
$\hat{r}$	a fitted reward	Section 15.3, Section 15.6
ν	Polyak rate of a target network	Section 15.5
$\hat{f}_\phi$	a learned transition model	Section 15.6

```
import gymnasium as gym
```

14.1.1 The Model

Figure 14.2 shows the lake: FrozenLake from the Gymnasium suite (Towers et al. 2024), which the tabular sections of this chapter share. The agent starts in the top-left cell and tries to reach the goal at the bottom right, which pays a reward of one. Four cells are holes, and entering a hole or the goal ends the attempt. On *slippery* ice a command moves the agent in the intended direction with probability $1/3$ and in each of the two perpendicular directions with probability $1/3$. A move off the edge of the grid leaves the agent where it is.

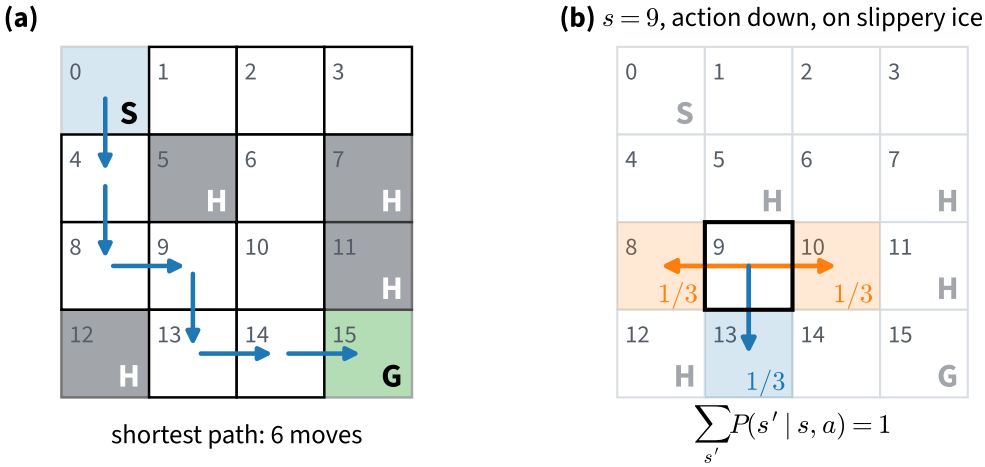


Figure 14.2: The lake. (a) S marks the start, G the goal, and the gray cells marked H are holes; each cell carries its state index. On calm ice the shortest path takes six moves. (b) One command on slippery ice: from state $s = 9$ the action *down* lands in cell 13, 8 or 10 with probability $1/3$ each. The intended move is one of three equally likely outcomes, and the probabilities sum to one.

Consider the agent in cell 9, three moves from the goal. Three of its four commands can slide it into the hole at cell 5, one time in three each. Only *down* cannot, but *down* also slides it sideways two times in three, into cell 8 or 10. Is the safety of *down* worth the steps it loses sliding sideways? To compare commands we need to know where each can lead, what the agent can do from there, and what it receives at the end.

14.1.1.1 States, Actions and the Transition Kernel

The *state* of the lake is the agent's cell. The set of states $\mathcal{S} = \{0, 1, \dots, 15\}$ numbers the cells row by row. The *actions* are the four commands *left*, *down*, *right* and *up*, encoded as 0 to 3; they form the set $\mathcal{A}$. On slippery ice an action determines a distribution over next states, and Gymnasium stores this distribution as a table that we can read directly:

```
env = gym.make('FrozenLake-v1', is_slippery=True)
LEFT, DOWN, RIGHT, UP = 0, 1, 2, 3
ARROWS = '<v>^'                          # one glyph per action, for printing
env.unwrapped.P[9][DOWN]
```

```
[(0.3333333333333333, 8, 0, False),
 (0.3333333333333333, 13, 0, False),
 (0.3333333333333333, 10, 0, False)]
```

The output lists three outcomes, each of the form (probability, next state, reward, terminated). Commanded *down* from cell 9, the agent reaches 8, 13 or 10 with probability $1/3$ each, as in Figure 14.2. In symbols, this table is the *transition kernel* $P(s' \mid s, a)$, the probability of reaching state s' from state s under action a. For each pair (s, a) the numbers $P(\cdot \mid s, a)$ form a distribution, so $\sum_{s' \in \mathcal{S}} P(s' \mid s, a) = 1$.

The fourth ingredient is the *reward*. The simulator pays one for the step that enters the goal and zero for every other step, so the reward of a step depends on where the step lands. We work with its average over the landing cell, the *expected reward* $r(s, a)$ of action a at state s; exercise 3 shows that the averaging changes no expected return. Together, the four objects form the MDP:

$$\text{MDP}: \quad (\mathcal{S}, \mathcal{A}, P, r). \tag{14.1}$$

The first three describe the environment. The reward describes the task, and whoever sets the task chooses it, a difference that the section on reward design turns on. Some treatments add the discount factor defined below to this tuple, or let the reward be random or depend on the next state. Our $r(s, a)$ then plays the role of the expected reward, and nothing in this chapter loses generality from the simpler form.

14.1.2 Return, Discount and Horizon

The agent starts at a state s_0 drawn from a *start-state distribution* μ_0, which for the lake puts all its mass on cell 0. At each step t the agent is at state s_t, takes action a_t, receives reward r_t, and moves to a next state $s_{t+1} \sim P(\cdot \mid s_t, a_t)$. The sequence of these quantities is a *trajectory*,

$$\tau = (s_0, a_0, r_0, s_1, a_1, r_1, s_2, a_2, r_2, \ldots).$$

The reward r_t in a trajectory is the realized one that the simulator pays; the expected reward $r(s_t, a_t)$ is its average given s_t and a_t.

A rule for choosing actions is called a *policy*; we write $\pi(a \mid s)$ for the probability that it picks action a in state s. The simplest one picks each of the four commands with probability $1/4$, whatever the state. We run it once from cell 0 and print every step:

```
rng = np.random.default_rng(8)
s, _ = env.reset(seed=8)
terminated = truncated = False
total, t = 0.0, 0
while not (terminated or truncated):
```

```
    a = int(rng.integers(4))
    s_next, r, terminated, truncated, _ = env.step(a)
    print(f"t={t:>2}  s={s:>2}  a={ARROWS[a]}  s'={s_next:>2}  r={r:.0f}")
    total, s, t = total + r, s_next, t + 1
print(f'terminated={terminated}, truncated={truncated}, total reward={total:.0f}')
```

```
t= 0  s= 0  a=>  s'= 0  r=0
t= 1  s= 0  a=v  s'= 0  r=0
t= 2  s= 0  a=<  s'= 4  r=0
t= 3  s= 4  a=^  s'= 4  r=0
t= 4  s= 4  a=<  s'= 4  r=0
t= 5  s= 4  a=v  s'= 8  r=0
t= 6  s= 8  a=>  s'= 9  r=0
t= 7  s= 9  a=^  s'=10  r=0
t= 8  s=10  a=>  s'=11  r=0
terminated=True, truncated=False, total reward=0
```

The transcript shows the kernel at work. At $t = 0$ the agent commands *right* and stays in cell 0, because the ice slid it into the wall above. At $t = 2$ it commands *left* and slides *down* to cell 4. At $t = 8$ it commands *right* from cell 10 and enters the hole at cell 11, which ends the attempt with `terminated=True`. No step entered the goal, so every reward is zero. To score a trajectory that does reach the goal, we must first decide how much a late reward counts.

14.1.2.1 Discounting and the Effective Horizon

Why not simply add the rewards? A trajectory can be infinitely long. The policy that always commands *up* never leaves the top row of the lake: *up* runs into the wall, and its two slips move along the row. With a positive reward on every step, the sum along such a trajectory grows without bound. A reward now is also usually worth more than the same reward later. Both concerns lead to a *discount factor* $0 \le \gamma < 1$, which weights the reward of step t by γ^t. The *return* of a trajectory is the discounted sum

$$R(\tau) = r_0 + \gamma r_1 + \gamma^2 r_2 + \cdots = \sum_{t=0}^{\infty} \gamma^t r_t. \tag{14.2}$$

With $\gamma = 0.95$ a reward one step away counts 0.95, and a reward twenty steps away counts about 0.36. The goal is to act so that the return is as large as possible on average, over the draws of the kernel, the start state and the agent's own choices. We maximize this *expected return* because it is what many independent attempts collect on average. For a finite MDP with $\gamma < 1$ the problem has a definite answer (Puterman 1994). There is one policy that is best from every start state at once, and it can be chosen deterministic and independent of time: one command per cell. So the best policy does not depend on where the agent starts, and a start-state distribution concentrated on one cell loses nothing.

The discount also controls how far ahead the agent looks. If every reward satisfies $|r_t| \le r_{\max}$, discounting keeps the return finite and bounds the part of it that lies beyond step k:

$$|R(\tau)| \le \frac{r_{\max}}{1-\gamma}, \qquad \Big|\sum_{t \ge k} \gamma^t r_t\Big| \le \frac{\gamma^k\, r_{\max}}{1-\gamma}.$$

The first bound makes the objective finite. The second says that rewards beyond step k can move the return by at most a fraction γ^k of its largest possible size. This suggests measuring the reach of the discount in steps, by the *effective horizon* $1/(1-\gamma)$. For γ near one, $\gamma^{1/(1-\gamma)} \approx e^{-1}$, so the weights fall by a factor of about e per horizon. For a constant reward stream, 95 percent of the discounted weight then lies within $\log(0.05)/\log(\gamma)$ steps, about three horizons. The table below lists both step counts for four discounts, together with the weight γ^5 that the goal's reward receives at the end of the six-move shortest path on calm ice:

```
print(f'{"gamma":>6} {"horizon 1/(1-gamma)":>20} {"t: gamma^t < 0.05":>18} '
      f'{"gamma^5":>8}')
for g in [0.5, 0.9, 0.95, 0.99]:
    t95 = int(np.ceil(np.log(0.05) / np.log(g)))
    print(f'{g:>6} {1 / (1 - g):>20.0f} {t95:>18} {g**5:>8.3f}')
```

gamma	horizon 1/(1-gamma)	t: gamma^t < 0.05	gamma^5
0.5	2	5	0.031
0.9	10	29	0.590
0.95	20	59	0.774
0.99	100	299	0.951

At $\gamma = 0.95$, the discount used throughout this chapter, the horizon is 20 steps, 95 percent of the weight lies within 59 steps, and a goal reached in six moves is worth $\gamma^5 \approx 0.774$. At $\gamma = 0.5$ the same goal is worth about 0.03. Figure 14.3 shows these quantities. The discount is part of the task, chosen by its designer: a small γ makes the agent short-sighted, but exercise 8 shows that it can make planning with an inaccurate model more reliable.

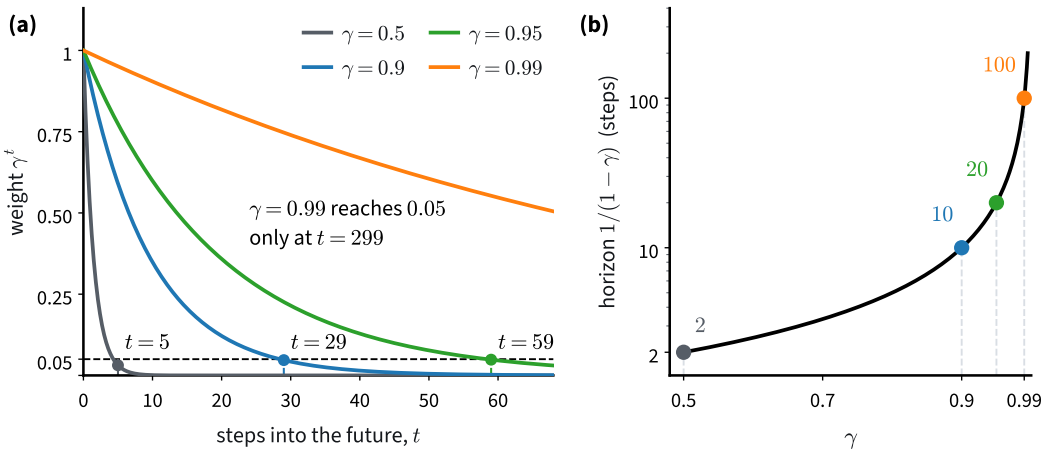


Figure 14.3: Discounting defines an effective horizon. (a) The weight γ^t of a reward t steps away falls below 0.05 at $t = 5$ for $\gamma = 0.5$, at $t = 29$ for $\gamma = 0.9$, at $t = 59$ for $\gamma = 0.95$, and only at $t = 299$ for $\gamma = 0.99$. (b) The horizon $1/(1-\gamma)$ on a logarithmic axis: two steps at $\gamma = 0.5$, twenty at $\gamma = 0.95$ and a hundred at $\gamma = 0.99$.

14.1.2.2 The Model as Arrays

Dynamic programming reads the kernel, the reward and the discount many times over, and every later tabular section checks its learner against those exact computations. The dictionary of tuples that Gymnasium exposes is awkward for that. We want P as one array indexed by (s, a, s') and r as one array indexed by (s, a), so that an average over next states becomes one matrix product. `TabularMDP` holds the three objects. Its `from_gym` method builds the arrays from the simulator's table: it adds up outcomes that lead to the same next state, and it averages the reward over the outcomes, which gives the expected reward $r(s, a)$. We save the class to the `d2l` library, since every tabular section of the chapter uses it:

```
class TabularMDP:
    """A finite MDP as dense arrays: P[s, a, s'] and r[s, a]."""
    def __init__(self, P, r, gamma):
        self.P, self.r, self.gamma = P, r, gamma
        self.num_states, self.num_actions = r.shape

    @classmethod
    def from_gym(cls, env, gamma):
        """Read the transition table Gymnasium exposes as env.unwrapped.P."""
        n_s, n_a = env.observation_space.n, env.action_space.n
        P, r = np.zeros((n_s, n_a, n_s)), np.zeros((n_s, n_a))
        for s, actions in env.unwrapped.P.items():
            for a, outcomes in actions.items():
                for p, s_next, reward, _ in outcomes:
                    P[s, a, s_next] += p      # several outcomes may share s'
                    r[s, a] += p * reward     # r(s,a) is the expected reward
        return cls(P, r, gamma)
```

We build the lake's model with $\gamma = 0.95$ and print the row of the command we read from the simulator above:

```
gamma = 0.95
mdp = TabularMDP.from_gym(env, gamma)
print(mdp.P[9, DOWN])
```

```
[0.         0.         0.         0.         0.         0.
 0.         0.         0.33333333 0.         0.33333333 0.
 0.         0.33333333 0.         0.        ]
```

The list of tuples has become a dense row of sixteen numbers: $1/3$ in columns 8, 10 and 13, zero elsewhere. Two checks, one per array, confirm the construction. Every row of P must sum to one, and the reward must be nonzero only where the goal can be entered:

```
print(f'largest |row sum - 1|: {np.abs(mdp.P.sum(axis=-1) - 1).max():.1e}')
for s, a in np.argwhere(mdp.r > 0):
    print(f'r(s={s}, a={ARROWS[a]}) = {mdp.r[s, a]:.3f}')
```

```
largest |row sum - 1|: 0.0e+00
r(s=14, a=v) = 0.333
r(s=14, a=>) = 0.333
r(s=14, a=^) = 0.333
```

Every row of P sums to one. The reward array is zero except in cell 14, the only frozen cell next to the goal. There *down*, *right* and *up* each reach the goal with probability $1/3$ and so have expected reward $1/3$, while none of the three outcomes of *left* is the goal. The lake pays its reward only on the step that enters the goal.

14.1.2.3 Episodes, Termination and Truncation

A state that ends the process is *terminal*; on the lake, the holes and the goal are terminal. A run from the start to a terminal state is an *episode*. The simulator stops an episode there and must be reset. Gymnasium's table, and so `TabularMDP`, stores each terminal state as an absorbing state whose actions lead back to itself with zero reward, so that every row of P is a distribution. The two descriptions give the same returns, since the absorbing state collects nothing more.

When every episode ends after finitely many steps, discounting is not needed to keep the return finite, and $\gamma = 1$ is allowed: the return is then the plain sum of the rewards of an episode with T steps. On the lake not every episode ends, since the always-*up* policy circles the top row forever, but $\gamma = 1$ is still safe there: rewards are nonnegative and a trajectory collects at most one. It is not safe in general. With a reward of $+0.05$ per step, the same always-*up* policy would have an infinite undiscounted return. Tasks that never end, such as controlling the heating of a building, therefore use $\gamma < 1$. Maximizing the long-run reward per step is the other common choice for such tasks; this chapter does not treat it.

Gymnasium separates two ways in which an episode can stop. `terminated=True` means that the process entered a terminal state: nothing more can happen, and the value of continuing is zero. `truncated=True` means that recording stopped, usually because of a time limit, while the agent stands in a state from which the process would go on. `gym.make` gives FrozenLake such a limit: after 100 steps the episode ends with `truncated=True`, even if the agent stands on frozen ice from where the goal can still be reached. Figure 14.4 shows both cases. The sum of rewards of a truncated episode is therefore not its return, since it misses everything after the cut. A method that estimates the value of continuing from the last state must treat them differently: the value is zero after termination, and after truncation it is whatever the last state is worth, its *value* $V(s)$, the expected return from s on, which Section 14.2 defines precisely. Every method in this chapter that estimates the value of continuing sets it to zero on `terminated` alone.

14.1.3 The Choice of State

The kernel $P(s' \mid s, a)$ conditions on the current state and action, and on nothing else. This is the *Markov property*: once the present state is known, the earlier history says nothing more about the next state. It is what keeps the model small. Because the future depends on the present state alone, how good it is to be in a state can be summarized by one number per state, whatever path led there. On the lake the property holds by construction. The simulator samples the next cell from the row `P[s][a]` that we read above and draws each slip afresh. Whether the property holds in general depends on what we call the state.

14.1.3.1 State Augmentation

Suppose the recorded state s_t is a vehicle's location and the action a_t is its acceleration. The next location also depends on the current velocity, which the current location alone does not

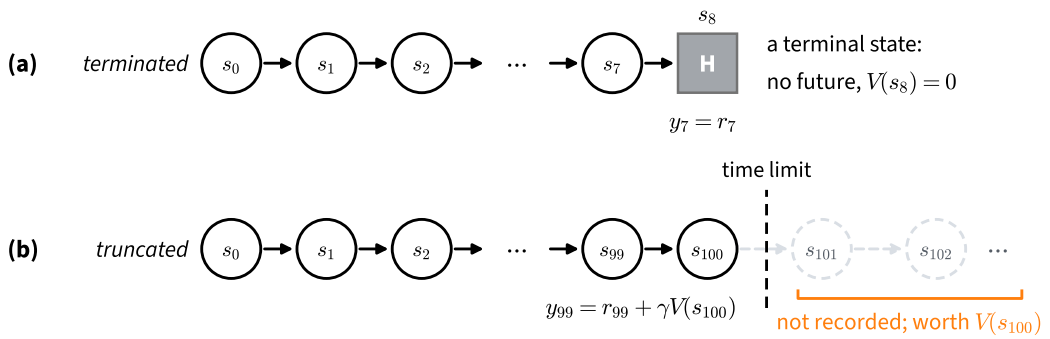


Figure 14.4: Two ways an episode stops. (a) Terminated: the episode enters the hole s_8, a terminal state. Nothing follows, so the value of continuing from it, $V(s_8)$, is zero, and the target for the last step, y_7, the return counted from that step, is the last reward r_7 alone. (b) Truncated: the 100-step time limit cuts the recording while the agent stands on frozen ice, from where the process would go on. The unrecorded future is worth $V(s_{100})$, the value of continuing from the last recorded state, which is not zero, so an estimate of the return from the last step, $y_{99} = r_{99} + \gamma V(s_{100})$, must include it.

reveal. Velocity can be inferred from two successive locations, so the next location depends on two past ones,

$$s_{t+1} = f(s_t, a_t, s_{t-1}), \tag{14.3}$$

and the location by itself is not Markov. Defining the state as the pair (location, velocity) resolves the problem: Newtonian dynamics determine the next location and velocity from the current pair and the applied acceleration. The Markov property is therefore a property of how we represent the state, not only of the physical system.

This construction is called *state augmentation*: the state is enlarged until it contains what is needed to predict the future. The enlarged state can be large, and the information it needs can be unknown, too large to store, or impossible to observe. A time limit that is part of the task is a simple case: with 100 steps allowed, the best command in a cell can depend on how many steps remain, and adding that count to the state restores the property (exercise 5).

14.1.3.2 Partial Observability

Augmentation requires the added variables to be observable. A poker player cannot see the opponents' cards. When the agent receives an observation o_t that reveals only part of the state, the problem is a *partially observable* MDP. An exact solution tracks the agent's *belief*, the posterior distribution of the hidden state given everything observed so far (exercise 7). In practice agents approximate it with memory. Early Atari agents stacked the last few frames, since one frame shows positions but not velocities. Current agents carry a recurrent state or attend over their whole history, as a language model does over its context. The rest of this chapter assumes that the agent observes a Markov state.

14.1.4 Reward Design

The states, actions and kernel describe the environment; the reward describes the task, and the designer chooses it. An optimizer maximizes whatever reward is written down, including any difference between that reward and what the designer meant (Krakovna et al. 2020). The lake shows how this happens in a few lines of arithmetic.

FrozenLake's reward is *sparse*: it is zero on every step except the one that enters the goal. An agent that has never reached the goal has seen nothing but zeros, so it is tempting to help it along. Let $\ell(s)$ be the Manhattan distance from cell s to the goal, the number of moves it would take if no hole were in the way, and pay a bonus of 0.3 for every move that gets closer:

$$\tilde{r}(s, a, s') = r(s, a) + 0.3\,\mathbf{1}\big(\ell(s') < \ell(s)\big). \tag{14.4}$$

The bonus seems to point the agent toward the goal, but it can be collected without ever finishing. Take the two cells next to the goal on the bottom row, 13 and 14, with $\ell = 2$ and $\ell = 1$, and suppose for the moment that the ice is calm. The move from 14 to 13 earns nothing and costs nothing, and the move back earns 0.3. From cell 14, entering the goal collects 1.3, the reward and the bonus, and ends the episode. Shuttling to 13 and back instead collects 0.3 on every second step, forever, which at $\gamma = 0.95$ is worth $0.3\gamma/(1-\gamma^2) \approx 2.9$. The flaw is that the bonus pays for moving closer but charges nothing for moving away, so it adds up to a positive amount around any loop. Randløv and Alstrøm (1998) met the same flaw in a simulated bicycle that was rewarded for progress toward a goal and learned to ride in circles.

A bonus that cannot be collected around a loop must charge for each move away what it pays for the move back. Bonuses of the form

$$\tilde{r}(s, a, s') = r(s, a) + \gamma\Phi(s') - \Phi(s), \tag{14.5}$$

for some function $\Phi : \mathcal{S} \to \mathbb{R}$ called a *potential*, have this property. Around a loop that returns to its start, the differences $\Phi(s') - \Phi(s)$ cancel, so apart from the effect of discounting a loop earns nothing. This construction is called *potential-based shaping*. With $\Phi(s) = -0.3\,\ell(s)$ it pays 0.315 for the move from 13 to 14 and charges 0.27 for the move back. Shuttling forever from cell 14 now collects 0.3 in total, while entering the goal still collects 1.3: the shaped reward keeps the goal ahead of the loop by exactly the one that the original reward pays. Such a bonus is still useful: a potential spreads the goal's reward over the route, which gives a learner a signal long before it first reaches the goal.

One condition remains: the potential must be zero at the terminal states. The simulator stops at a terminal state, so a nonzero potential there would leave a final term that depends on which terminal state the episode ends in, and when. Such a term can change which behavior is best. Our $\Phi = -0.3\,\ell$ is zero at the goal but not at the holes, so it must be set to zero there. What the approach bonus does to the best policy on slippery ice is computed in Section 14.2, which also proves that the potential form leaves the best policy unchanged.

14.1.5 Two Special Cases

Two kinds of MDP recur in this chapter, because each removes one difficulty of the lake. When there is only one state, each round consists of choosing an action and observing a reward, and no action affects the future. This is the *multi-armed bandit*, named after a row of

slot machines whose payout rates a player must discover by playing; it isolates the question of which actions to try, which Section 14.4 studies. The second case keeps the future but removes the randomness of the transitions, and it describes a language model writing a response.

14.1.5.1 Text Generation as an MDP

A language model that writes a response one token at a time fits the MDP with simple dynamics. The state is the prompt together with the tokens written so far, the action is the next token, and the transition appends that token to the state with probability one. The reward arrives once, when the response is complete: a checker called a *verifier* compares the response with a known answer, or a learned model scores it. Two properties set this MDP apart from the lake. The transitions are deterministic, so all randomness comes from the policy. And the state is the whole prefix, so no state is ever visited twice, and the states form a tree rooted at the prompt.

With deterministic transitions, a single reward at the end, and no revisits, the expected return depends only on the probability that the policy gives to each complete response. Nothing is lost by treating the whole response as one action chosen at the prompt. The MDP then collapses to one step: the prompt x, drawn from a distribution μ_0 over prompts, is the state; a complete response y is the action; and the verifier's score $r(x, y)$ is the reward. This is why methods for language models can score and update whole responses. Section 15.8 returns to the token-level view.

14.1.5.2 The Verifier Task

We make the one-step view concrete with an instance small enough to compute exactly, the second running example of this chapter. It has four arithmetic prompts and nine candidate responses shared by all of them. The verifier pays one when a response equals the prompt's answer exactly, and zero otherwise. Two of the prompts, `8*8` and `9*9`, have no correct candidate, so no policy can score more than 0.5 on average over the prompts. The ninth candidate, "one of 4, 12, 64 or 81", lists every answer at once. The exact-match verifier rejects this *hedge*, but a lenient checker that accepts any response containing the answer would pay it on every prompt, the problem of reward design in another form.

The policy stands in for a pretrained language model. It is a table θ of preferences with one row per prompt and one column per response, turned into probabilities by a softmax over each row, $\pi(y \mid x) \propto \exp \theta_{xy}$, as in softmax regression. The reference table θ_{ref} is zero except for the hedge column, which is -4. A reference response is therefore one of the eight ordinary candidates with probability about $1/8$ each, and the hedge with probability about 0.2 percent. The class `VerifierTask` collects these pieces. Its method `verify(x, y)` is the checker, and its attribute `R` holds the checker's rewards as a table, one row per prompt and one column per response. Its method `policy(theta)` applies the softmax to each row of a preference table. Its method `success(pi)` is the expected score of one sampled response, averaged over the four prompts: the expected return of this one-step MDP, with μ_0 uniform over the prompts. An optional `reward` argument scores the policy under another checker instead. Training changes the table; here we only score the reference:

```
class VerifierTask:
    """Four prompts, nine candidate responses shared by all of them, an
```

```
    exact-match verifier, and a reference policy that rarely hedges."""
    prompts = ['2+2', '3*4', '8*8', '9*9']
    answers = ['4', '12', '64', '81']
    resp = ['1', '3', '4', '7', '12', '19', '23', '35',
            'one of 4, 12, 64 or 81']          # the last one is the hedge

    def __init__(self):
        self.theta_ref = np.zeros((len(self.prompts), len(self.resp)))
        self.theta_ref[:, -1] = -4.0           # the reference rarely hedges
        self.pi_ref = self.policy(self.theta_ref)
        self.R = self.reward_table(self.verify)   # R[x, y] = r(x, y)

    def verify(self, x, y):
        """The verifier: reward 1 iff response y is prompt x's answer."""
        return float(self.resp[y] == self.answers[x])

    def reward_table(self, reward):
        """reward(x, y) for every prompt x (rows) and response y (columns)."""
        return np.array([[reward(x, y) for y in range(len(self.resp))]
                         for x in range(len(self.prompts))])

    @staticmethod
    def policy(theta):
        """pi(y | x): a softmax over each prompt's row of preferences."""
        z = np.exp(theta - theta.max(1, keepdims=True))
        return z / z.sum(1, keepdims=True)

    def success(self, pi, reward=None):
        """E r(x, y), y ~ pi(. | x), averaged over the prompts; r = verify
        unless another reward function is given."""
        R = self.R if reward is None else self.reward_table(reward)
        return (pi * R).sum(1).mean()

task = VerifierTask()
print(f'success of a response sampled from the reference: '
      f'{task.success(task.pi_ref):.3f}')
```

```
success of a response sampled from the reference: 0.062
```

A response sampled from the reference passes the verifier with probability 0.062. On each of the two answerable prompts the correct candidate has probability about $1/8$, and the other two prompts contribute nothing, so the average is about $1/16$. Training on this task has to close the gap between 0.062 and the ceiling of 0.5.

We can now write a decision problem down: the lake as two arrays and a discount, the verifier task as a table of scores, and for each the return that a trajectory earns. A model of this kind can be queried at any state and action, which is what *model-based* methods use; *model-free* methods see only the transitions that the agent's own experience produces. The question posed at cell 9 is still open. Comparing *down* with *right* requires the return that each command can lead to, and that depends on what the agent does in every cell it may reach next. Section 14.2 computes these values for all cells at once.

14.1.6 Summary

A Markov decision process writes a problem of acting over time as states, actions, a transition kernel and a reward, $(\mathcal{S}, \mathcal{A}, P, r)$, under the assumption that the present state carries everything about the past that matters for the future. The agent maximizes the expected discounted return; the discount γ keeps the return finite and sets an effective horizon of $1/(1-\gamma)$ steps, with weights that fall by a factor of about e per horizon when γ is near one. Terminal states end the process, while truncation only stops the recording, so the value of continuing is zero only after termination. When the obvious state is not Markov, augmenting it restores the property if the missing information can be observed. The reward is a design choice: a bonus that can be collected around a loop changes what the agent does, while a potential-based bonus cannot be collected that way. FrozenLake and the four-prompt verifier task, a token MDP collapsed to one step, are the running examples of the chapter.

14.1.7 Exercises

1. **What the state must contain.** In each system below the obvious state is not Markov. Name the smallest addition that restores the Markov property, and say whether the agent can observe it.

    1. A delivery robot whose state is its location, when every move drains its battery by an amount that depends on the terrain and on the load it carries, and the robot stops once the battery is empty.
    2. A recommender whose state is the user's profile, when the user's interest in a topic falls with the number of recent recommendations of that topic.
    3. A chess program whose state is the arrangement of the pieces on the board, when the game ends in a draw as soon as a position occurs for the third time, or as soon as fifty moves by each player pass without a capture or a pawn move.

    In which of the three systems can the addition grow with the length of the episode, and what keeps it bounded there?

2. code **Episodes from the kernel.** The example trajectory τ of this section draws every action uniformly at random with `rng.integers(4)`. Under this policy π the next state has the distribution

$$P_\pi(s' \mid s) = \frac{1}{4} \sum_{a \in \mathcal{A}} P(s' \mid s, a),$$

    which is `mdp.P.mean(axis=1)`.

    1. `from_gym` accumulates probabilities with `+=` because several of Gymnasium's outcomes for one state-action pair can lead to the same next state. Find every pair at which this happens, and explain from Figure 14.2 why these pairs and no others. If the line that fills `P` assigned with `=` instead, which rows of `mdp.P` would no longer sum to one, and what would they sum to?
    2. Let d_t^π be the distribution of s_t, starting from $d_0^\pi = \mu_0$, the point mass on state 0, so that $d_{t+1}^\pi(s') = \sum_s d_t^\pi(s) P_\pi(s' \mid s)$. Plot, for $t \le 30$, the probability that the agent has reached the goal, the probability that it has fallen into a hole, and the probability that its episode is still running. What are the limits as $t \to \infty$?

3. Ignoring the time limit of `env`, show that the mean episode length equals $\sum_{t\geq 0} \Pr(s_t \text{ is not terminal})$. Run 10,000 episodes of `env` with the random policy, and compare the fraction of episodes that reach the goal and the mean episode length with the values that d_t^π predicts. The 100-step limit of `env` ends a random episode with probability below 10^{-8}, so it does not affect the comparison.

3. **Expected rewards.** Gymnasium attaches its reward to the transition: entering the goal pays one. `from_gym` replaces this reward $\rho(s, a, s')$ by its expectation $r(s, a) = \sum_{s'} P(s' \mid s, a)\, \rho(s, a, s')$.
   1. Show that for every policy and every start state the expected discounted return is the same whether step t pays $\rho(s_t, a_t, s_{t+1})$ or $r(s_t, a_t)$. Extend the argument to a reward that is drawn at each visit of (s, a), independently of everything else, from a distribution with mean $r(s, a)$.
   2. Under the convention $r(s, a)$, which FrozenLake episodes have a positive return? For the uniformly random policy of this section, estimate from 10,000 episodes the probability of a positive return under each of the two conventions. Which properties of the return does the model Equation 14.1 therefore determine, and which does it not?

4. **Discount as termination.** The discounted return Equation 14.2 weights the reward of step t by γ^t.
   1. This section locates 95 percent of the discounted weight within $\log(0.05)/\log(\gamma)$ steps and calls this about three effective horizons. Show that for every fraction $p \in (0, 1)$

$$\frac{\ln(1/p)}{\ln(1/\gamma)} \leq \frac{\ln(1/p)}{1-\gamma}, \tag{14.6}$$

      and that the ratio of the two sides tends to one as $\gamma \to 1$. Use the inequality to explain the step counts of Figure 14.3 at $\gamma = 0.5$, 0.9 and 0.99.
   2. Modify an MDP so that after every reward the process moves, with probability $1-\gamma$, to an absorbing state with zero reward, and otherwise moves according to P. Show that the expected undiscounted return of every policy in the modified process equals its expected discounted return in the original, and that the expected number of rewards collected is $1/(1-\gamma)$. Read the arrays of `TabularMDP` in these terms: what do the row sums of γP mean?

5. **Time limits.** `gym.make` wraps FrozenLake in a time limit: after 100 steps the episode ends with `truncated=True` (Section 14.1.2.3). Suppose that the limit is part of the task: the objective is the probability of entering the goal within 100 steps. Model the end of the limit as a move to an absorbing state. Show that the distribution of the next state then depends on the number h of steps that remain, so that the location alone is not a Markov state, whereas the pair (s, h) is (Pardo et al. 2018). Why can the best action at a location depend on h?

6. **Actions without consequences.** A multi-armed bandit has a single state, so no action can influence the future.
   1. Suppose more generally that the kernel does not depend on the action, $P(s' \mid s, a) = P(s' \mid s)$ for all a. Show that for every γ the policy that picks an action

in $\arg\max_a r(s,a)$ at every state maximizes the expected discounted return from every start state.

2. Construct an MDP with two states and two actions in which this greedy policy is optimal for small γ and not for large γ, and find the value of γ at which the optimal action changes.

7. **Belief states.** ▯ In a partially observable MDP the agent receives an observation o_t, drawn with probability $O(o_t \mid s_t)$, instead of the state s_t. Its *belief* $\beta_t(s) = \Pr(s_t = s \mid o_0, a_0, \dots, a_{t-1}, o_t)$ is the posterior over the hidden state given everything observed so far (Kaelbling et al. 1998).

   1. Using the Markov property of the hidden states, derive the update

$$\beta_{t+1}(s') \propto O(o_{t+1} \mid s') \sum_{s \in \mathcal{S}} P(s' \mid s, a_t)\, \beta_t(s). \tag{14.7}$$

   2. Show that the belief is a Markov state: given β_t and a_t, the distribution of β_{t+1} and the expected reward do not depend on earlier observations. What are the state space and the kernel of the resulting MDP over beliefs?

   3. In a T-maze adapted from Bakker (2001), the agent observes a cue at the start, *left* or *right* with probability $1/2$ each, then moves one cell per step through N corridor cells that all produce the same observation, and then reaches a junction. Turning toward the cue there gives reward one, and turning away gives zero. Show that a policy that acts on the k most recent observations succeeds with probability at most $1/2$ unless $k \geq N + 2$. What is the belief at the junction, and how much memory suffices to act optimally? Why does stacking a few frames work for Pong but not here?

8. [extended] **Planning horizon and model error.** ▯▯ The discount γ of the objective is fixed by the task, but a planner may compute its policy with a smaller *planning discount* γ'. Jiang et al. (2015) found that when the model is estimated from data, $\gamma' < \gamma$ can yield a better policy than $\gamma' = \gamma$. Let $V^{\pi}_{\gamma}(s)$ be the expected discounted return from s under policy π and discount γ. Part (b) uses the value iteration and exact policy evaluation of Section 14.2 (`d2l.value_iteration` and `TabularMDP.policy_value`).

   1. Let rewards lie in $[0, r_{\max}]$ and let $\gamma' \leq \gamma$. Show that for every policy π and state s

$$0 \leq V^{\pi}_{\gamma}(s) - V^{\pi}_{\gamma'}(s) \leq \frac{(\gamma - \gamma')\, r_{\max}}{(1-\gamma)(1-\gamma')}, \tag{14.8}$$

   and conclude that on the true model the policy that is optimal for γ' loses at most this amount under γ.

   2. Reproduce their experiment at small scale with `TabularMDP`. Draw 100 random MDPs with 10 states and 2 actions: for each state-action pair, choose 5 next states uniformly without replacement, give them weights drawn uniformly from $[0,1]$, and normalize; draw each $r(s,a)$ uniformly from $[0,1]$. For each MDP and each $n \in \{2, 5, 10, 20\}$, build 20 estimated kernels $\widehat{P}$ from n next states sampled per pair, keeping the true r. Compute the optimal policy of each estimated model for $\gamma' \in \{0, 0.1, \dots, 0.9, 0.99\}$, evaluate it on the true model at $\gamma = 0.99$, and

average the loss $V_\gamma^*(s) - V_\gamma^\pi(s)$ over states, kernels and MDPs, where V_γ^* is the value of the optimal policy of the true model. Value iteration needs about $\log(10^{-8})/\log\gamma'$ sweeps, about 2,000 at $\gamma' = 0.99$ and far fewer below; with that schedule the experiment takes a few minutes of CPU time. Plot the loss against γ' for each n.

3. Which γ' minimizes the loss for each n, and how does it move with n? Explain the shape of the curves with two sources of loss, the horizon bias bounded in this problem and the error of $\hat{P}$, and say which of the two grows with γ'.

14.2 Dynamic Programming

The previous section wrote the frozen lake down as two arrays and a discount, and left a question open. In cell 9, *down* is the only command that cannot slide the agent into the hole next to it, but it slides the agent sideways two times in three. Is that safety worth the lost moves? On calm ice there would be nothing to decide: the shortest path to the goal takes six moves, and following it is best. On slippery ice the shortest path runs past the holes, and a command that gains a cell can, on a slip, end the episode. Which command is best cannot be read off the next cell. It depends on how much reward the agent can still expect from every cell it may land in, if it keeps acting well from there.

Richard Bellman studied such multistage decisions in the 1950s and found that this score, the *value* of a state, satisfies a local equation (Bellman 1957b). The value of a cell is the reward of one move plus the discounted value of the cell that the move leads to, averaged over the slips. Values are defined by the whole future, yet each one is tied only to the values of its neighbors, so all of them can be computed together by repeating a one-step look-ahead. Bellman called the method *dynamic programming*. Its answers are the exact ones that the learning methods of later sections try to approach, and on a small problem they let us check those methods.

This section defines values for a fixed policy and for the best one, derives Bellman's equations, and shows that repeating the look-ahead converges, with a rule that says when to stop. On the slippery lake the policy it finds reaches the goal about sixteen times as often as the shortest path, and in four cells it walks away from the goal; we compute why. At the end we return to the approach bonus of the previous section and compute what it does to the best policy. Everything here reads the known transition table P.

```
import gymnasium as gym
```

14.2.1 Values and the Bellman Equations

14.2.1.1 Policies and Values

Section 14.1 called any rule for choosing actions a policy. In general a *policy* $\pi(a \mid s)$ is a conditional distribution over the actions $a \in \mathcal{A}$ given the state $s \in \mathcal{S}$: at some cell of the lake the probabilities of *left*, *down*, *right* and *up* might be $(0.4, 0.2, 0.1, 0.3)$. We write $\pi(\cdot \mid s)$ for the whole distribution at s. A *deterministic* policy puts all its probability on one action, which we write $\pi(s)$.

Fix a policy. An agent that starts from s_0 draws $a_t \sim \pi(\cdot \mid s_t)$, moves to $s_{t+1} \sim P(\cdot \mid$

$s_t, a_t)$, and collects the discounted return $\sum_t \gamma^t r(s_t, a_t)$ of Section 14.1. Different runs give different trajectories, so we score the start state by the average return,

$$V^\pi(s_0) = \mathbb{E}_{a_t \sim \pi(\cdot|s_t),\ s_{t+1} \sim P(\cdot|s_t, a_t)} \Big[\sum_{t=0}^{\infty} \gamma^t r(s_t, a_t) \Big]. \tag{14.9}$$

This is the *value function* of π. The start state is arbitrary, so V^π assigns a number to every state: the reward an agent that follows π from there can expect to collect. On the lake the only reward is the one for entering the goal, so $V^\pi(s)$ is the probability of reaching the goal from s, with each success weighted by γ^t for the step t on which it enters the goal.

The question at cell 9 compares commands, while V^π averages over them. The *action value* pins the first action and follows the policy afterwards:

$$Q^\pi(s_0, a_0) = r(s_0, a_0) + \mathbb{E}_{a_t \sim \pi(\cdot|s_t),\ s_{t+1} \sim P(\cdot|s_t, a_t)} \Big[\sum_{t=1}^{\infty} \gamma^t r(s_t, a_t) \Big]. \tag{14.10}$$

The sum starts at $t = 1$ because the first reward is fixed by a_0: take the given action for one step, then follow the policy forever. Averaging the action value over the policy's own first action gives back the value,

$$V^\pi(s) = \sum_{a \in \mathcal{A}} \pi(a \mid s)\, Q^\pi(s, a) \quad \text{for all } s \in \mathcal{S}. \tag{14.11}$$

Thus $V^\pi(s)$ is a weighted average of the numbers $Q^\pi(s, \cdot)$ and never exceeds the largest of them: $\max_a Q^\pi(s, a) \geq V^\pi(s)$, with equality only if the policy puts all its probability on actions that attain the maximum. Every improvement step in this section starts from this inequality.

14.2.1.2 The Expectation Form

The values are defined by infinite sums, but the Markov property ties each one to its neighbors. Split a trajectory after its first transition. The first part contains the action a_0, the reward $r(s_0, a_0)$ and the next state s_1; the rest is a trajectory that starts at s_1. By the Markov property the expected return of the rest depends on the past only through s_1, and it equals $V^\pi(s_1)$. The value at s_0 is therefore the expected immediate reward plus the discounted expected value of the next state:

$$V^\pi(s_0) = \mathbb{E}_{a_0 \sim \pi(\cdot|s_0)} \Big[r(s_0, a_0) + \gamma\, \mathbb{E}_{s_1 \sim P(\cdot|s_0, a_0)} \Big[V^\pi(s_1) \Big] \Big]. \tag{14.12}$$

The outer expectation averages over the policy's choice of a_0; the inner one averages over the environment's next state s_1 given that action. For finite state and action spaces, the expectations are sums:

$$V^\pi(s) = \sum_{a \in \mathcal{A}} \pi(a \mid s) \Big[r(s, a) + \gamma \sum_{s' \in \mathcal{S}} P(s' \mid s, a) V^\pi(s') \Big]; \text{ for all } s \in \mathcal{S}. \tag{14.13}$$

Conditioning also on the first action gives the corresponding equation for the action value:

$$Q^\pi(s,a) = r(s,a) + \gamma \sum_{s' \in \mathcal{S}} P(s' \mid s,a) \sum_{a' \in \mathcal{A}} \pi(a' \mid s')\, Q^\pi(s',a'); \;\; \text{for all } s \in \mathcal{S}, a \in \mathcal{A}. \tag{14.14}$$

These are the *Bellman expectation equations*. For a fixed policy, Equation 14.13 is a system of $|\mathcal{S}|$ linear equations in the $|\mathcal{S}|$ unknowns $V^\pi(s)$: the infinite horizon has disappeared into the recursion.

One cell of the lake shows what kind of system this is. Take a policy that commands *down* in cell 14, next to the goal. *Down* runs into the bottom wall and leaves the agent in cell 14, and the two slips carry it left to cell 13 or right into the goal, which pays one. Each outcome has probability $1/3$, and the goal, a terminal state, has value zero, so Equation 14.13 at cell 14 reads

$$V^\pi(14) = \frac{1}{3} + \frac{\gamma}{3}[V^\pi(13) + V^\pi(14)].$$

The unknown $V^\pi(14)$ appears on both sides, because a slip can leave the agent where it was, and $V^\pi(13)$ depends in turn on its own neighbors, cell 14 among them. In a game tree the values could be filled in from the leaves back to the root, one at a time. The slippery lake has loops, and its equations must be solved together: as one linear system, or by starting from a guess and applying the right-hand side over and over. Both ways appear below.

Every right-hand side above evaluates the same bracket for one state-action pair: the immediate reward plus the discounted expected value of the next state. This one-step look-ahead is called a *backup*, because it carries values back from the successors to the state. On the arrays of `TabularMDP` from Section 14.1, the backups of all state-action pairs are one matrix product, which contracts the last axis of `P[s, a, s']` against a value vector `V`. We add it to the class as the method `backup`, which returns the array of brackets indexed by (s,a):

```
@d2l.add_to_class(d2l.TabularMDP)
def backup(self, V):
    """Q(s, a) = r(s, a) + gamma * sum_{s'} P(s'|s, a) V(s') for every
    state-action pair: the bracket of the Bellman equations."""
    return self.r + self.gamma * self.P @ V
```

14.2.1.3 The Optimality Equation

An *optimal* policy maximizes the value. In principle the best policy could depend on the start state, look at the whole history of the episode, or randomize. For a discounted finite MDP none of this is needed, as Section 14.1 stated:

Proposition (a uniformly optimal policy exists). *In a finite MDP with discount $0 \le \gamma < 1$ there is a deterministic, stationary policy π^* that is optimal at every state simultaneously: $V^{\pi^*}(s) \ge V^\pi(s)$ for all $s \in \mathcal{S}$ and all policies π, including stochastic policies and policies that depend on the history* (Puterman 1994).

The value of a policy that depends on the history is defined by the same expected return. We write $V^* \equiv V^{\pi^*}$ and $Q^* \equiv Q^{\pi^*}$, and ask which equation V^* satisfies. Two comparisons

give the answer. First, the expectation equation of π^* says that $V^*(s)$ equals the bracket $r(s,a) + \gamma \sum_{s'} P(s' \mid s,a) V^*(s')$ of the action $a = \pi^*(s)$, so $V^*(s)$ cannot exceed the largest bracket. Second, a policy that takes some action a at s once and follows π^* afterwards has value equal to the bracket of a; by the proposition no policy beats π^*, so no bracket exceeds $V^*(s)$. Together, the two comparisons say that $V^*(s)$ is the largest bracket:

$$V^*(s) = \max_{a \in \mathcal{A}} \Big[r(s,a) + \gamma \sum_{s' \in \mathcal{S}} P(s' \mid s,a)\, V^*(s') \Big]; \text{ for all } s \in \mathcal{S}. \tag{14.15}$$

This is the *Bellman optimality equation* (Bellman 1952, 1957b). In words, the value of a state under optimal play is the value of the best first move followed by optimal play from wherever that move leads. Unlike the expectation form, the equation mentions no policy: it is an equation in the unknown table V^*, and it is nonlinear because of the maximum. The first comparison also shows that the action $\pi^*(s)$ attains the maximum. Given V^*, an optimal action is therefore found by one look-ahead with the model:

$$\pi^*(s) = \operatorname*{argmax}_{a \in \mathcal{A}} \Big[r(s,a) + \gamma \sum_{s' \in \mathcal{S}} P(s' \mid s,a)\, V^*(s') \Big]. \tag{14.16}$$

The action-value version of Equation 14.15 follows by inserting $V^*(s') = \max_{a'} Q^*(s',a')$ into the bracket:

$$Q^*(s,a) = r(s,a) + \gamma \sum_{s' \in \mathcal{S}} P(s' \mid s,a) \max_{a' \in \mathcal{A}} Q^*(s',a'). \tag{14.17}$$

The two forms differ in what extraction needs. From Q^* the optimal action is $\pi^*(s) = \operatorname{argmax}_a Q^*(s,a)$, with no model, whereas extraction from V^* through Equation 14.16 needs P and r. The difference matters as soon as the model is unknown. When the model is known, the one-step look-ahead of Equation 14.16 can also be extended to many steps. Programs that play board games search many moves ahead, with the rules of the game as the model (Silver et al. 2018) or with a model they learn (Schrittwieser et al. 2020).

14.2.1.4 Backup Diagrams

The four equations share one structure: the value at a root is assembled from the values one step below it. Figure 14.5 draws them side by side. The last diagram shows what a learner must do without the model: it cannot average over the environment's next states, so it uses the one transition it observed.

14.2.2 Value Iteration

We load the slippery lake of Section 14.1 as a `TabularMDP` with $\gamma = 0.95$, together with one arrow glyph per action for printing:

```
gamma = 0.95
ARROWS = '<v>^'                    # left, down, right, up: actions 0 to 3
env = gym.make('FrozenLake-v1', is_slippery=True)
mdp = d2l.TabularMDP.from_gym(env, gamma)
```

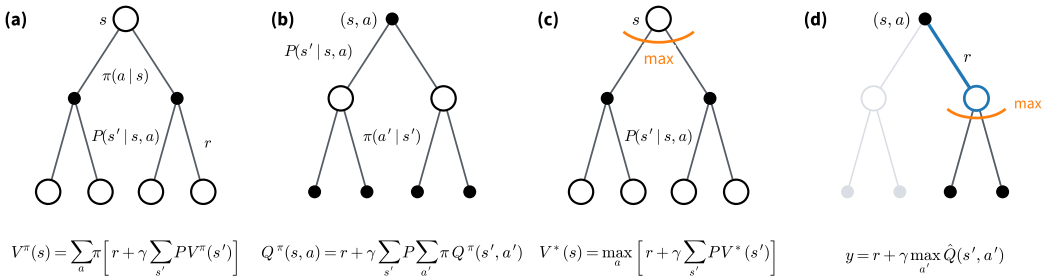


Figure 14.5: Backup diagrams. Open circles are states, filled dots are state-action pairs, and each diagram shows how the value at the root is assembled from the values one step below it. Left to right: (a) the value function averages over the policy's actions and the environment's next states; (b) the action-value function fixes the first action and averages afterwards; (c) the optimality operator replaces the average over actions by a maximum, drawn as the arc; (d) a sampled backup cannot take the environment's average, so it uses the single observed transition, in blue, and keeps the maximum at the next state.

14.2.2.1 The Algorithm

The optimality equation defines V^* only implicitly. To compute it, read the right-hand side of Equation 14.15 as a map: the *Bellman optimality operator* $\mathcal{T}$ takes any table $V : \mathcal{S} \to \mathbb{R}$ and returns the table of best brackets,

$$(\mathcal{T}V)(s) = \max_{a \in \mathcal{A}} \Big[r(s, a) + \gamma \sum_{s' \in \mathcal{S}} P(s' \mid s, a)\, V(s') \Big]. \tag{14.18}$$

The optimality equation says $V^* = \mathcal{T}V^*$: the optimal value function is a *fixed point* of $\mathcal{T}$. The simplest way to look for a fixed point is to apply the map over and over. Start from $V_0 = 0$ and sweep

$$V_{k+1}(s) = \max_{a \in \mathcal{A}} \Big[r(s, a) + \gamma \sum_{s' \in \mathcal{S}} P(s' \mid s, a)\, V_k(s') \Big]; \text{ for all } s \in \mathcal{S}, \tag{14.19}$$

that is, $V_{k+1} = \mathcal{T}V_k$. This is *value iteration* (Bellman 1957b). Each sweep performs one backup per state-action pair and keeps the best one per state. In code the sweep is one line, because backup computes all the brackets at once. The function returns every iterate, so that we can watch the values move:

```
def value_iteration(mdp, num_iters):
    """Sweep V <- max_a backup(V); return the whole history of iterates."""
    V, history = np.zeros(mdp.num_states), []
    for _ in range(num_iters):
        V = mdp.backup(V).max(axis=1)
        history.append(V)
    return np.array(history)
```

14.2.2.2 Calm Ice

The smallest case is the lake without slips, where every command does what it says. The goal's reward is paid on the move into the goal, so a cell one move from the goal has value 1. A cell two moves away has value γ, because its best move reaches a cell of value 1 one step later. In general a cell whose shortest route around the holes takes j moves has value $V^*(s) = \gamma^{j-1}$. Value iteration from $V_0 = 0$ finds these values one ring at a time. After sweep k, the cells within k moves of the goal have nonzero values, each already final, and the others are still zero. Figure 14.6 shows the first six sweeps.

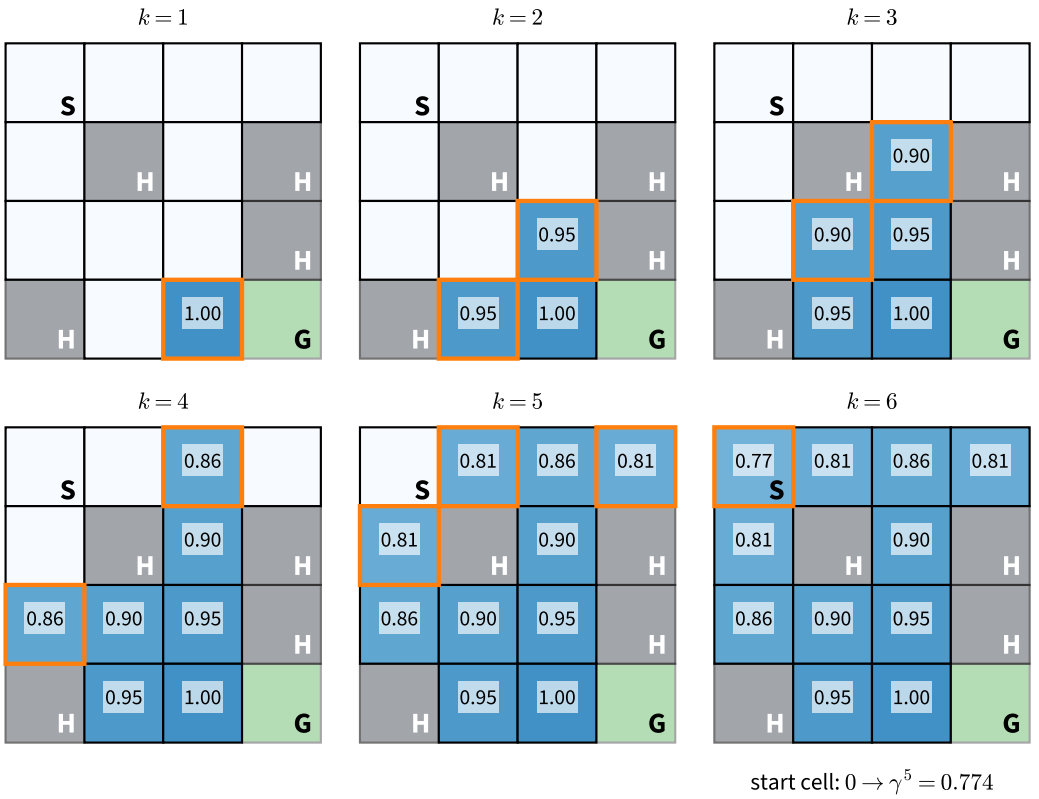


Figure 14.6: Value iteration on calm FrozenLake, whose values have a closed form. Sweep k reaches the cells within k moves of the goal, the orange outline marking the newly reached ones, and a cell j moves from the goal has value γ^{j-1}. The start cell is six moves away, so its estimate stays at zero for five sweeps and equals $\gamma^5 = 0.774$ from the sixth on.

The cell below runs ten sweeps on the calm lake, counts the cells with a nonzero value after each sweep, and extracts the greedy policy `shortest` by Equation 14.16:

```
calm = d2l.TabularMDP.from_gym(
    gym.make('FrozenLake-v1', is_slippery=False), gamma)
history_calm = value_iteration(calm, num_iters=10)
V_calm = history_calm[-1]
shortest = calm.backup(V_calm).argmax(axis=1)  # ties go to the lower action
```

```
print(f'cells with nonzero value after each sweep: '
      f'{(history_calm > 0).sum(axis=1)}')
print(f'V*(s0) on calm ice: {V_calm[0]:.4f};  gamma^5 = {gamma ** 5:.4f}')
```

```
cells with nonzero value after each sweep: [ 1  3  5  7 10 11 11 11 11 11]
V*(s0) on calm ice: 0.7738;  gamma^5 = 0.7738
```

The count grows 1, 3, 5, 7, 10, 11 and then stays at 11, the number of frozen cells other than the goal. After six sweeps every value is final, and the start cell holds $\gamma^5 = 0.774$. On calm ice, value iteration is a shortest-path algorithm: values travel back from the goal one move per sweep. Its greedy policy `shortest` moves along a shortest route from every cell. At cells 0 and 9 two moves are equally short, and `argmax` picks the first of them, *down*.

Slips change one thing. A slip can carry the agent back to a cell it came from, or leave it in place, as at cell 14 above, so the values feed back into each other. No finite number of sweeps then reaches them, and we must ask whether the iteration converges at all, how fast, and when to stop.

14.2.2.3 Convergence via Contraction

All three answers rest on one property of $\mathcal{T}$: it brings any two tables closer together. We measure the distance between tables in the sup norm $\|V\|_\infty = \max_s |V(s)|$, the largest disagreement at any state.

Proposition (the Bellman operator is a γ-contraction). *For any two tables $V, V' : \mathcal{S} \to \mathbb{R}$,*

$$\|\mathcal{T}V - \mathcal{T}V'\|_\infty \le \gamma \, \|V - V'\|_\infty .$$

Proof. Fix s and write $f(a)$ and $g(a)$ for the bracketed terms in $(\mathcal{T}V)(s)$ and $(\mathcal{T}V')(s)$. For every a, $|f(a) - g(a)| = \gamma \, |\sum_{s'} P(s' \mid s, a)(V(s') - V'(s'))| \le \gamma \|V - V'\|_\infty$, because the probabilities are nonnegative and sum to one. The difference between two maxima is bounded by the largest difference of their inputs, $|\max_a f(a) - \max_a g(a)| \le \max_a |f(a) - g(a)|$, so $|(\mathcal{T}V)(s) - (\mathcal{T}V')(s)| \le \gamma \|V - V'\|_\infty$ for every s. ■

Thus one sweep shrinks the largest disagreement between any two tables by at least the factor γ, whatever the tables are. Applied to V_k and to V^*, which $\mathcal{T}$ leaves in place, it says that every sweep shrinks the error of value iteration by at least γ. Figure 14.7 shows this on an MDP with two states, where a table is a point in the plane.

Four consequences follow. The first is that the fixed point is unique. If V and V' are both fixed points, then $\|V - V'\|_\infty = \|\mathcal{T}V - \mathcal{T}V'\|_\infty \le \gamma \|V - V'\|_\infty$, which for $\gamma < 1$ forces $V = V'$. So V^* is the only solution of the optimality equation.

The second is that value iteration converges from any start at a geometric rate. Write $e_k = \|V_k - V^*\|_\infty$ for the error after k sweeps. Then

$$e_k = \|\mathcal{T}V_{k-1} - \mathcal{T}V^*\|_\infty \le \gamma \, e_{k-1} \le \cdots \le \gamma^k e_0 , \tag{14.20}$$

so the error falls below 10^{-6} after at most $\log(e_0/10^{-6}) / \log(1/\gamma)$ sweeps. For γ near one the denominator is about $1 - \gamma$, and the sweep count grows in proportion to the effective horizon $1/(1-\gamma)$ of Section 14.1.

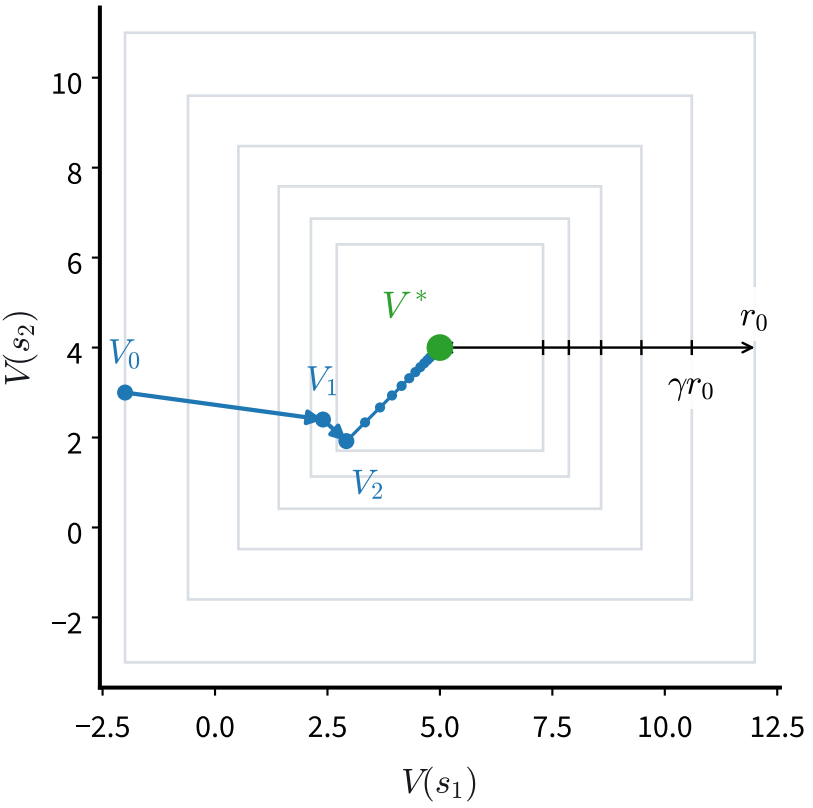


Figure 14.7: Value iteration converges because the Bellman operator $\mathcal{T}$ is a contraction in the sup norm. The MDP has two states; in each the agent may stay or switch, staying in s_1 pays 1, every other choice pays 0, and $\gamma = 0.8$, so $V^* = (5, 4)$. The squares are the sup-norm balls of radius $\gamma^k r_0$ around V^*, where $r_0 = \|V_0 - V^*\|_\infty = 7$ for the start $V_0 = (-2, 3)$, and sweep k lands inside the k-th square.

The third is that the iterates certify their own accuracy. The error e_k cannot be observed, since V^* is what we are computing, but the change between consecutive iterates can, and it bounds the error (exercise 2 derives the bound):

$$\|V_k - V^*\|_\infty \le \frac{\gamma}{1-\gamma}\,\|V_k - V_{k-1}\|_\infty. \tag{14.21}$$

The right-hand side depends only on the last two iterates, so value iteration can stop as soon as it falls below the target.

The fourth concerns policies. Replacing the maximum in $\mathcal{T}$ by the average under a fixed policy π gives the operator $\mathcal{T}^\pi$, which maps a table V to the right-hand side of the expectation equation Equation 14.13 with V in place of V^π. The proof above, with an average in place of the maximum, shows that $\mathcal{T}^\pi$ is a γ-contraction too, so V^π is its unique fixed point (Puterman 1994; Bertsekas 2025). Now let π be greedy with respect to V^*, as in Equation 14.16. Then $\mathcal{T}^\pi V^* = \mathcal{T}V^* = V^*$, so V^* is the fixed point of $\mathcal{T}^\pi$, which is V^π. Every greedy policy of V^* is therefore optimal, whichever way ties between maximizing actions are broken, and extracting a policy with `argmax` is safe.

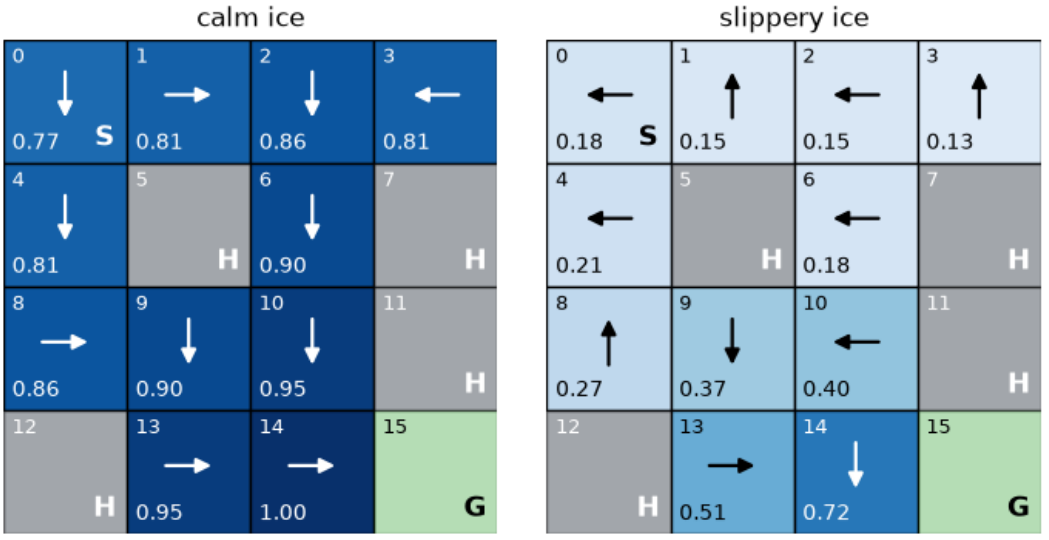
All four consequences need $\gamma < 1$. At $\gamma = 1$ the proof shows only that $\mathcal{T}$ does not increase distances, and both uniqueness and convergence can fail (exercise 3). The always-*up* policy of Section 14.1, which never ends its episode, shows how: with a reward of 0.05 per step, its undiscounted values would grow by 0.05 with every sweep and never converge. Problems in which every policy ends its episodes with probability one, called stochastic shortest-path problems, keep a unique solution at $\gamma = 1$ once the values of the terminal states are fixed at zero (Bertsekas 2025).

14.2.2.4 Slippery Ice

We now run value iteration on the slippery lake. After 1,000 sweeps the guarantee $\gamma^{1000} e_0$ is below 10^{-22}, so we take the last iterate as V^* and extract its greedy policy π^* by Equation 14.16. The cell checks the equation of cell 14 with the computed values, prints the action values $Q^*(9, \cdot)$ of the question in cell 9, and draws the values and greedy commands of both lakes:

```
history = value_iteration(mdp, num_iters=1000)
V_star = history[-1]
Q_star = mdp.backup(V_star)
pi_star = Q_star.argmax(axis=1)
print(f'V*(s0) = {V_star[0]:.4f}, V*(13) = {V_star[13]:.4f}')
print(f'state 14: V* = {V_star[14]:.4f};  1/3 + gamma (V*(13) + V*(14)) / 3 = '
      f'{1 / 3 + gamma * (V_star[13] + V_star[14]) / 3:.4f}')
print(f'cell 9: Q*(9, a) for a = {", ".join(ARROWS)}: {Q_star[9].round(3)}')
d2l.show_grid(env.unwrapped.desc, [V_calm, V_star], [shortest, pi_star],
              titles=['calm ice', 'slippery ice'])
```

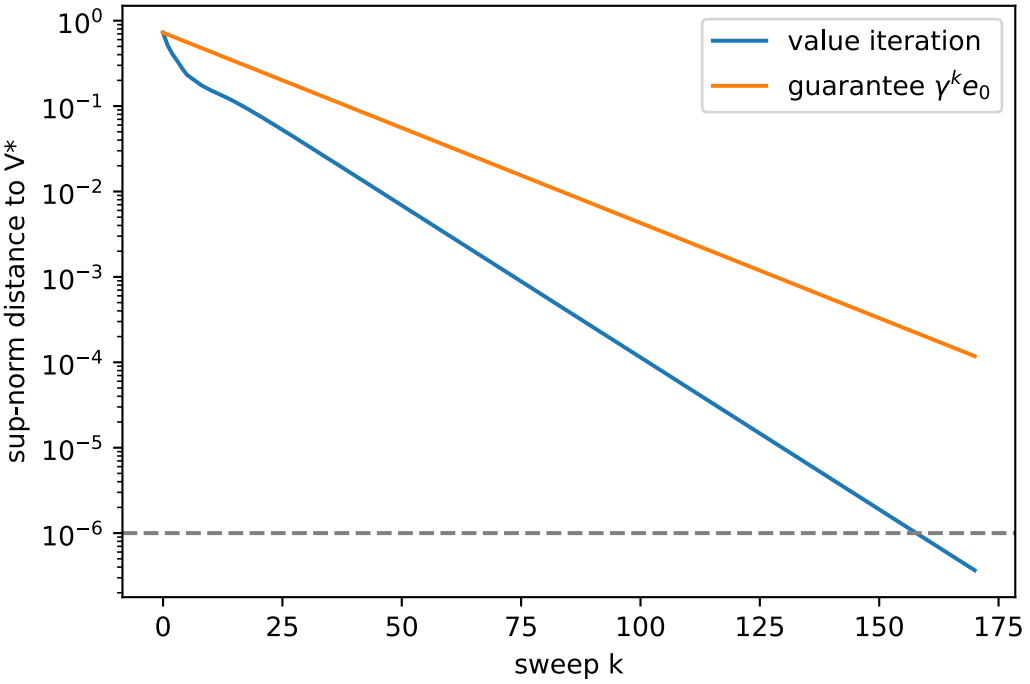
```
V*(s0) = 0.1805, V*(13) = 0.5090
state 14: V* = 0.7237;  1/3 + gamma (V*(13) + V*(14)) / 3 = 0.7237
cell 9: Q*(9, a) for a = <, v, >, ^: [0.247 0.375 0.289 0.213]
```



The equation of cell 14 holds with $V^*(13) = 0.509$: $\frac{1}{3} + 0.95\,(0.509 + 0.724)/3 = 0.724$, the same number on both sides. The question of cell 9 has its answer. Commanding *down* is worth 0.375, against 0.289 for *right*, the better of the commands that risk the hole: the safety of *down* is worth the sideways slips. Compared with calm ice, the slips lower every value. The start cell falls from 0.774 to 0.180, and even cell 14, beside the goal, is worth 0.724 rather than 1. The commands change as well. In four cells the slippery optimum points away from the goal, and in several others into a wall; Section 14.2.4 explains each of them.

How fast did the sweeps converge? The cell below plots the error e_k of the first 170 sweeps against the guarantee $\gamma^k e_0$, with the tolerance 10^{-6} as a dashed line, and asserts that the guarantee is never violated:

```
err = np.concatenate([[np.abs(V_star).max()],
                      np.abs(history - V_star).max(axis=1)])
k = np.arange(171)
assert (err[k] <= err[0] * gamma ** k + 1e-12).all()
d2l.plot_curves({'value iteration': err[k],
                 r'guarantee $\gamma^k e_0$': err[0] * gamma ** k},
                xlabel='sweep k', ylabel='sup-norm distance to V*',
                yscale='log', reference=1e-6)
```



On a logarithmic scale a geometric decay is a straight line. The guarantee falls by $\gamma = 0.95$ per sweep; from $e_0 = V^*(14) = 0.724$ it reaches 10^{-6} at sweep 264. The measured error falls faster, by about 0.92 per sweep, and crosses the dashed line at sweep 158. The guarantee holds for every MDP with this discount, so neither the number of cells nor their distance from the goal enters it (exercise 7 explains the faster rate).

14.2.2.5 Stopping Rules

An implementation cannot watch the true error, since it does not know V^*. It must stop on something it can see. The cell compares two observable rules with the truth: stop when the sweep-to-sweep change $\|V_k - V_{k-1}\|_\infty$ falls below 10^{-6}, or when the certificate Equation 14.21 does:

```
gap = np.abs(np.diff(history, axis=0)).max(axis=1)
true_err = np.abs(history[1:] - V_star).max(axis=1)
certified = gamma / (1 - gamma) * gap
assert (true_err <= certified + 1e-12).all()
for name, e in [('sweep-to-sweep change', gap),
                ('certified error bound', certified),
                ('distance to V*', true_err)]:
    print(f'{name} first below 1e-6 at sweep {np.argmax(e <= 1e-6) + 2}')
print(f'distance to V* when the change first falls below 1e-6: '
      f'{true_err[np.argmax(gap <= 1e-6)]:.1e}')
k_cert = np.argmax(certified <= 1e-6) + 2
```

```
sweep-to-sweep change first below 1e-6 at sweep 128
certified error bound first below 1e-6 at sweep 164
```

```
distance to V* first below 1e-6 at sweep 158
distance to V* when the change first falls below 1e-6: 1.1e-05
```

The rule on the change stops at sweep 128. There the true error is still 1.1×10^{-5}, more than eleven times the tolerance, and nothing the rule can see reveals this. The certificate stops at sweep 164, six sweeps after the true error first fell below 10^{-6} at sweep 158. It is conservative, but only slightly.

14.2.3 Policy Evaluation and Policy Iteration

Value iteration improves the policy implicitly, at every sweep. A second family of methods separates the two jobs: measure how good a given policy is, then improve it. Measuring is also useful on its own, since it scores any policy, including ones we did not optimize.

14.2.3.1 Policy Evaluation

Replace the maximum in the sweep by the policy's own average, that is, apply $\mathcal{T}^{\pi}$ instead of $\mathcal{T}$:

$$V_{k+1}^{\pi}(s) = \sum_{a \in \mathcal{A}} \pi(a \mid s) \Big[r(s,a) + \gamma \sum_{s' \in \mathcal{S}} P(s' \mid s,a)\, V_k^{\pi}(s') \Big]; \text{ for all } s \in \mathcal{S}. \tag{14.22}$$

This is *policy evaluation*. It converges to V^{π} from any start because $\mathcal{T}^{\pi}$ is a γ-contraction. The code is the value-iteration sweep with the maximum replaced by the average under `pi[s, a]`; we apply it to the uniformly random policy:

```python
def policy_evaluation(mdp, pi, num_iters):
    """Same sweep with the max replaced by an average under pi(a|s)."""
    V, history = np.zeros(mdp.num_states), []
    for _ in range(num_iters):
        V = (pi * mdp.backup(V)).sum(axis=1)
        history.append(V)
    return np.array(history)

uniform = np.ones((mdp.num_states, mdp.num_actions)) / mdp.num_actions
print(f'V(s0) for the uniformly random policy: '
      f'{policy_evaluation(mdp, uniform, 400)[-1][0]:.4f}')
```

```
V(s0) for the uniformly random policy: 0.0078
```

The uniformly random policy has start value 0.0078, about four percent of the optimum 0.180: a walker that picks its commands at random rarely reaches the goal, and late when it does.

Without the maximum the equations are linear, and a table admits a direct solution. Write $P_{\pi}(s' \mid s) = \sum_a \pi(a \mid s) P(s' \mid s,a)$ for the kernel under π, the probability of moving from s to s' in one step when the action is drawn from π, and $r_{\pi}(s) = \sum_a \pi(a \mid s) r(s,a)$ for its expected reward. Then Equation 14.13 reads $V^{\pi} = r_{\pi} + \gamma P_{\pi} V^{\pi}$, whose solution is

$$V^{\pi} = (I - \gamma P_{\pi})^{-1} r_{\pi}. \tag{14.23}$$

The matrix $I - \gamma P_\pi$ is invertible for $\gamma < 1$, because γP_π shrinks every vector in the sup norm; exercise 1 reads its inverse as discounted visit counts. For a table, one linear solve gives V^π exactly. The sweeps do the same work one contraction at a time, and they are the form that extends to problems too large to solve directly. We add the solve to `TabularMDP`:

```
@d2l.add_to_class(d2l.TabularMDP)
def policy_value(self, pi):
    """V^pi = (I - gamma P_pi)^{-1} r_pi exactly, for pi[s, a] the
    probability of action a at state s (one-hot rows: deterministic).
    Needs gamma < 1 here: from_gym's terminal states loop on themselves."""
    P_pi = np.einsum('sa,sat->st', pi, self.P)
    r_pi = (pi * self.r).sum(axis=1)
    return np.linalg.solve(np.eye(self.num_states) - self.gamma * P_pi,
                           r_pi)

print(f'V(s0) for the uniformly random policy, solved exactly: '
      f'{mdp.policy_value(uniform)[0]:.4f}')
```

```
V(s0) for the uniformly random policy, solved exactly: 0.0078
```

Both methods give 0.0078. The `einsum` forms P_π by averaging the kernel's rows over the actions with the policy's probabilities. The solve needs $\gamma < 1$ on these arrays: `from_gym` stores each terminal state as a state whose actions lead back to itself, so at $\gamma = 1$ the matrix $I - P_\pi$ has a zero row there and is singular. Quantities at $\gamma = 1$, such as probabilities of reaching the goal, therefore use the sweeps.

14.2.3.2 Policy Iteration and Generalized Policy Iteration

Evaluation and the greedy step together suggest a different algorithm, due to Howard (1960): evaluate the current policy exactly, act greedily with respect to its values, and repeat. The greedy step rests on the inequality $\max_a Q^\pi(s,a) \geq V^\pi(s)$ from the start of the section, applied at every state at once.

Proposition (policy improvement). *Let π be any policy and let π' be greedy with respect to V^π, that is, $\pi'(s) = \operatorname{argmax}_a \big[r(s,a) + \gamma \sum_{s'} P(s' \mid s,a) V^\pi(s')\big]$. Then $V^{\pi'}(s) \geq V^\pi(s)$ at every state, and if equality holds everywhere, then both policies are optimal.*

Proof. Greediness says that one step of π' followed by the values of π is at least as good as one step of π: $\mathcal{T}^{\pi'} V^\pi = \mathcal{T} V^\pi \geq \mathcal{T}^\pi V^\pi = V^\pi$. The operator $\mathcal{T}^{\pi'}$ is monotone, since its coefficients are probabilities: $V \leq U$ at every state implies $\mathcal{T}^{\pi'} V \leq \mathcal{T}^{\pi'} U$. Applying it repeatedly gives $V^\pi \leq \mathcal{T}^{\pi'} V^\pi \leq (\mathcal{T}^{\pi'})^2 V^\pi \leq \cdots$, and the sequence converges to the unique fixed point of $\mathcal{T}^{\pi'}$, which is $V^{\pi'}$. If $V^{\pi'} = V^\pi$, the first line reads $\mathcal{T} V^\pi = V^\pi$, so V^π solves the optimality equation; its only solution is V^*, so π is optimal. ■

Each round of *policy iteration* therefore either raises the value at some state or leaves every value unchanged, and in the second case the policy is already optimal. The values never decrease, so no policy recurs before the optimum is reached, and a finite MDP has finitely many deterministic policies: policy iteration ends after finitely many rounds (Puterman 1994). The test for the end is the proposition's equality condition, that the greedy step gains nothing at any state. Testing instead whether the greedy policy has stopped changing can fail with ties, which the slippery lake has: at cell 6, *left* and *right* have the same action value, and rounding can flip `argmax` between them from one round to the next. The loop below

starts from the uniform policy, evaluates each policy with `policy_value`, and stops when no action beats the current value by more than rounding error:

```
pi, num_outer = uniform, 0
while True:
    V = mdp.policy_value(pi)                          # evaluate pi exactly
    Q = mdp.backup(V)
    num_outer += 1
    if (Q.max(axis=1) <= V + 1e-12).all():            # greedy gains nothing:
        break                                         # pi is optimal
    pi = np.eye(mdp.num_actions)[Q.argmax(axis=1)]    # improve
assert np.abs(V - V_star).max() < 1e-12
```

The assertion checks that policy iteration and value iteration agree on V^* to twelve decimal places. The next cell counts the work each did. A backup is one evaluation of the bracket for one state-action pair, so a sweep of value iteration costs $|\mathcal{S}||\mathcal{A}| = 64$ backups:

```
ok = [(mdp.backup(V_k).argmax(axis=1) == pi_star).all() for V_k in history]
assert all(ok[np.argmax(ok):])     # once optimal, the greedy policy stays so
first = mdp.backup(mdp.policy_value(uniform)).argmax(axis=1)
per_sweep = mdp.num_states * mdp.num_actions          # backups in one sweep
print(f'policy iteration: {num_outer} exact solves; the first greedy step '
      f'gives pi*: {(first == pi_star).all()}')
print(f'value iteration: certified at sweep {k_cert} '
      f'({k_cert * per_sweep:,} backups)')
print(f'its greedy policy already equals pi* from sweep {np.argmax(ok) + 1} '
      f'on ({(np.argmax(ok) + 1) * per_sweep:,} backups)')
```

```
policy iteration: 2 exact solves; the first greedy step gives pi*: True
value iteration: certified at sweep 164 (10,496 backups)
its greedy policy already equals pi* from sweep 14 on (896 backups)
```

Policy iteration needed two exact solves. The greedy policy of the uniform policy's values is already π^*, and the second round only confirms that the greedy step gains nothing. Value iteration needed 10,496 backups to certify its values, but its greedy policy was already π^* after 14 sweeps, or 896 backups. A policy depends only on which action value is largest, so it can settle long before the values do. Each round of policy iteration solves a linear system in 16 unknowns, which is cheap here; with many states, the cubic cost of a solve is what makes sweeps preferable (exercise 12 compares the costs).

Evaluation and improvement need not alternate in complete phases. *Generalized policy iteration* is the name for any scheme that interleaves steps that move the values toward V^π with steps that move the policy toward the greedy one (Sutton and Barto 2018). Policy iteration completes each evaluation before it improves; value iteration improves after a single sweep of evaluation. Methods that learn from experience take smaller and noisier steps of the same two kinds, computed from sampled transitions instead of the model. Figure 14.8 draws four such schemes.

14.2.4 The Optimal Policy on Slippery Ice

The values say that π^* is the best policy. How much better is it than the obvious alternative, the shortest path of calm ice, and why does it point away from the goal in four cells? We

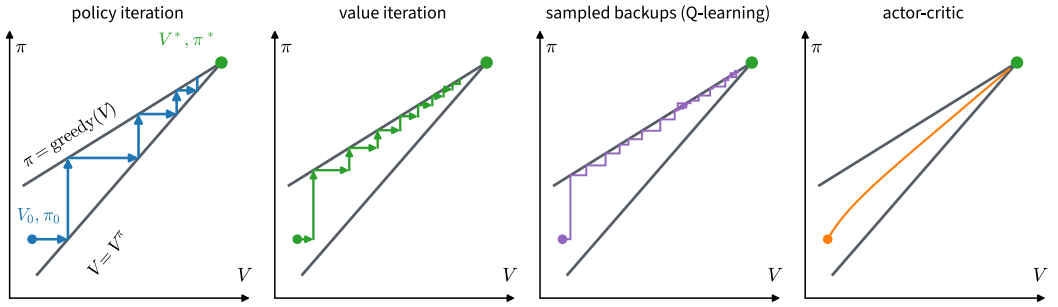


Figure 14.8: Generalized policy iteration. The line $V = V^\pi$ holds the value estimates that are exact for their policy, the line $\pi = \text{greedy}(V)$ the policies that are greedy for their estimate, and both hold only at (V^*, π^*). From the same start, policy iteration evaluates all the way to the first line before each improvement, value iteration takes one sweep toward it, sampled backups take short noisy steps, and methods that update a policy and its value estimate together move both coordinates a little at once. The axes stand for the spaces of value functions and policies; the drawing is schematic.

answer both questions exactly from the model, and check the answers by simulation.

14.2.4.1 Monte Carlo Evaluation

A check that does not use the model runs the policy in the simulator and averages the returns. For this purpose we represent a policy as a function `policy(obs, rng)` from an observation to an action, which draws from the random generator `rng` if the policy is stochastic. The function `evaluate` returns the mean discounted return over a number of episodes:

```
def evaluate(env, policy, num_episodes, gamma=1.0, rng=None):
    """Mean discounted return of an acting policy over Monte Carlo episodes.

    A policy is any function `policy(obs, rng) -> action`; a deterministic
    policy ignores `rng`, a stochastic one draws from it."""
    total = 0.0
    for _ in range(num_episodes):
        obs, done, discount = env.reset()[0], False, 1.0
        while not done:
            obs, reward, terminated, truncated, _ = env.step(policy(obs, rng))
            done = terminated or truncated
            total, discount = total + discount * reward, discount * gamma
    return total / num_episodes
```

The variable `discount` holds γ^t for the current step, and an episode ends on either `terminated` or `truncated`. With the default $\gamma = 1$ and the lake's single reward for entering the goal, the mean return is the fraction of episodes that reach the goal within the simulator's 100-step limit.

14.2.4.2 Success Rates

The same probability can be computed from the model. At $\gamma = 1$, k sweeps of policy evaluation from $V_0 = 0$ add up the expected rewards of the first k steps. The only reward is the one for entering the goal, so the k-th iterate at s_0 is the probability of reaching the goal within k steps, and $k = 100$ matches the simulator's limit. We build the model with $\gamma = 1$ as `lake1` and compute the probability for π^*, for `shortest` and for the uniform policy, each with a 2,000-episode Monte Carlo estimate beside it:

```
lake1 = d2l.TabularMDP.from_gym(env, 1.0)    # gamma = 1: V(s0) = P(goal)
rng = np.random.default_rng(0)
env.reset(seed=0)
for name, p in [('slip-aware optimum', np.eye(4)[pi_star]),
                ('shortest-path policy', np.eye(4)[shortest]),
                ('uniformly random policy', uniform)]:
    exact = policy_evaluation(lake1, p, 100)[-1][0]   # within 100 steps
    mc = evaluate(env, lambda s, rng: int(rng.choice(4, p=p[s])), 2000,
                  rng=rng)
    print(f'{name:>23}: goal within 100 steps {exact:.3f} exactly, '
          f'{mc:.3f} in 2000 episodes')
```

```
    slip-aware optimum: goal within 100 steps 0.730 exactly, 0.736 in 2000 episodes
  shortest-path policy: goal within 100 steps 0.045 exactly, 0.046 in 2000 episodes
uniformly random policy: goal within 100 steps 0.014 exactly, 0.010 in 2000 episodes
```

The optimal policy reaches the goal within 100 steps with probability 0.730, the shortest-path policy with probability 0.045, about one sixteenth as often, and the random walker with probability 0.014. Each Monte Carlo estimate differs from its exact value by no more than sampling noise explains: the standard error of 2,000 episodes is about one percentage point at 0.73 and a quarter of a point at 0.014. The time limit costs π^* a little, because its cautious commands make for long episodes and some that would reach the goal after more than 100 steps are cut off.

14.2.4.3 Commands That Turn Away from the Goal

Why does the shortest path fail so badly? We look at each of the eleven frozen cells where the agent acts and ask two things: where the command of π^* points, and how likely each policy's command is to move the agent into a hole on the next step. The direction is read from the calm-ice values: a command moves toward the goal if its calm-ice successor has a larger value `V_calm`, into a wall if the successor is the cell itself, and away from the goal otherwise; a move into a hole or the goal counts as away, since both have value zero on calm ice. The array `risk[s, a]` holds the one-step hole probability $\sum_{s' \text{ a hole}} P(s' \mid s, a)$:

```
desc = env.unwrapped.desc.flatten()
hole, end = desc == b'H', np.isin(desc, [b'H', b'G'])
risk = mdp.P[:, :, hole].sum(axis=-1)       # risk[s, a] = P(hole next | s, a)
move = calm.P[np.arange(16), pi_star] @ V_calm - V_calm   # on calm ice
for s in np.flatnonzero(~end):
    kind = ('toward the goal' if move[s] > 1e-9 else
            'into a wall' if move[s] > -1e-9 else 'away from the goal')
    print(f's={s:>2}: pi* {ARROWS[pi_star[s]]} {kind:<18}  hole risk: pi* '
```

```
            f'{risk[s, pi_star[s]]:.2f}, shortest {risk[s, shortest[s]]:.2f}, '
            f'least {risk[s].min():.2f}')
```

```
s= 0: pi* < into a wall         hole risk: pi* 0.00, shortest 0.00, least 0.00
s= 1: pi* ^ into a wall         hole risk: pi* 0.00, shortest 0.33, least 0.00
s= 2: pi* < away from the goal  hole risk: pi* 0.00, shortest 0.00, least 0.00
s= 3: pi* ^ into a wall         hole risk: pi* 0.00, shortest 0.33, least 0.00
s= 4: pi* < into a wall         hole risk: pi* 0.00, shortest 0.33, least 0.00
s= 6: pi* < away from the goal  hole risk: pi* 0.33, shortest 0.67, least 0.33
s= 8: pi* ^ away from the goal  hole risk: pi* 0.00, shortest 0.33, least 0.00
s= 9: pi* v toward the goal     hole risk: pi* 0.00, shortest 0.00, least 0.00
s=10: pi* < away from the goal  hole risk: pi* 0.00, shortest 0.33, least 0.00
s=13: pi* > toward the goal     hole risk: pi* 0.00, shortest 0.00, least 0.00
s=14: pi* v into a wall         hole risk: pi* 0.00, shortest 0.00, least 0.00
```

The optimal policy moves toward the goal in only two cells, 9 and 13. In five cells, 0, 1, 3, 4 and 14, it commands into a wall. Such a command turns the intended move into staying put and leaves the two slips as the moves that happen. At cell 4, for instance, *left* slides the agent up or down along the wall and never right into the hole at cell 5. At cell 14, *down* and *right* both reach the goal one time in three and stay in cell 14 one time in three. The third outcome is cell 13 for *down* and cell 10 for *right*, and cell 10 is worth less (0.40 against 0.51 in the grid above).

In four cells, 2, 6, 8 and 10, the optimal policy points away from the goal. At cells 8 and 10 the commands toward the goal can lead into a hole, 12 below cell 8 and 11 beside cell 10; *up* at 8 and *left* at 10 cannot, and *left* at 10 still slides the agent down to cell 14 one time in three. At cell 6 every command risks a hole, since holes lie on both sides. *Left* and *right* risk one third, *up* and *down* two thirds, and π^* takes one of the two least risky commands, which tie; `argmax` picks *left*, and *right* would point toward the goal. It commands the move into hole 5, which happens one time in three, because both of its slips land on safe cells, while both slips of *up* and *down* land in holes. At cell 2 no command risks a hole, and *left* wins because it cannot slip the agent into the corner cell 3, the least valuable cell of the top row, from which every route passes back through cell 2.

Over all eleven cells the risk of the optimal command is the least available, and it is zero everywhere except at cell 6. The shortest-path policy takes a risk of one third at cells 1, 3, 4, 8 and 10 and of two thirds at cell 6. These one-step risks do not yet say where episodes end, because that also depends on how often each cell is visited. For a deterministic policy π, let P_π be the kernel under π with the rows of the terminal states set to zero, so that the process stops there. Both policies studied here end every episode with probability one, so $I - P_\pi$ is invertible, and the expected numbers of visits to the states, starting from $s_0 = 0$, form the row of $(I - P_\pi)^{-1}$ that belongs to s_0, which the function `falls` obtains by solving the transposed system. Visits times the hole risk of the command in each cell give the probability that the episode ends in a hole from that cell:

```
def falls(mdp, pi, risk, end):
    """P(the episode falls into a hole from s), for each s, without a time
    limit: expected visits to s before the end, times the hole risk at s."""
    S, I = np.arange(mdp.num_states), np.eye(mdp.num_states)
    P_pi = mdp.P[S, pi] * ~end[:, None]               # stop at the end
    visits = np.linalg.solve(I - P_pi.T, I[0])        # starting from s0 = 0
```

```
    return visits * risk[S, pi] * ~end

for name, p in [('pi*', pi_star), ('shortest', shortest)]:
    f = falls(mdp, p, risk, end)
    print(f'{name:>8}: P(falls into a hole) {f.sum():.2f}, from ' +
          ', '.join(f'{s}: {f[s]:.2f}' for s in np.argsort(-f) if f[s] > 0))
```

```
    pi*: P(falls into a hole) 0.22, from 6: 0.22
shortest: P(falls into a hole) 0.96, from 4: 0.31, 1: 0.31, 8: 0.13, 6: 0.08, 3: 0.06, 10: 0.05
```

Without a time limit the optimal policy falls into a hole with probability 0.22, all of it at cell 6, and reaches the goal in the other 78 percent of its episodes; the 100-step limit lowers this to the 0.730 above. The shortest-path policy falls in 96 percent of its episodes. About a third of the falls start from cell 4, on its route, and another third from cell 1, which is not on the route at all. The first command, *down* at cell 0, slips the agent right to cell 1 one time in three, and there `shortest` commands *right*, which slips down into the hole at cell 5 one time in three. The rest of the falls start from cells 8, 6, 3 and 10. The shortest path fails less on its own route than in the cells that slips carry it to, and the optimal policy avoids the holes in every cell where some command can.

14.2.5 Reward Shaping and the Optimal Policy

Section 14.1 proposed helping a learner on the sparse lake with a bonus of 0.3 for every move that gets closer to the goal, in the Manhattan distance $\ell(s)$, and showed with calm-ice arithmetic that the bonus pays for going in circles. With value iteration and exact policy evaluation we can now compute what the bonus does to the best policy on slippery ice, and prove that the potential form proposed there leaves the best policy unchanged.

14.2.5.1 An Approach Bonus

Averaged over the next state, the bonus adds 0.3 times the probability of moving closer to the expected reward:

$$\tilde{r}(s,a) = r(s,a) + 0.3 \sum_{s'} P(s' \mid s,a)\, \mathbf{1}(\ell(s') < \ell(s)).$$

The cell builds this reward for bonuses of 0.3, 0.1 and 0, finds the optimal policy of each shaped MDP by value iteration, and evaluates that policy under the original reward with `policy_value`. The row with bonus 0 is the original problem:

```
row, col = np.divmod(np.arange(16), 4)
dist = abs(row - 3) + abs(col - 3)     # l(s): grid steps to the goal at (3, 3)
closer = dist < dist[:, None, None]    # closer[s, :, s'] = 1 iff l(s') < l(s)

for bonus in [0.3, 0.1, 0.0]:
    shaped = d2l.TabularMDP(mdp.P, mdp.r + bonus * (mdp.P * closer).sum(-1),
                            gamma)
    V_shaped = value_iteration(shaped, num_iters=500)[-1]
    pi_shaped = shaped.backup(V_shaped).argmax(axis=1)
    V_true = mdp.policy_value(np.eye(mdp.num_actions)[pi_shaped])
```

```
    print(f'bonus {bonus}: at s=14 goes {ARROWS[pi_shaped[14]]}, shaped V(0) '
          f'{V_shaped[0]:.2f}, V(13) {V_shaped[13]:.2f}, V(14) '
          f'{V_shaped[14]:.2f}; true V(0) {V_true[0]:.3f}')
```

```
bonus 0.3: at s=14 goes <, shaped V(0) 2.15, V(13) 1.71, V(14) 1.52; true V(0) 0.000
bonus 0.1: at s=14 goes v, shaped V(0) 0.79, V(13) 0.87, V(14) 0.94; true V(0) 0.115
bonus 0.0: at s=14 goes v, shaped V(0) 0.18, V(13) 0.51, V(14) 0.72; true V(0) 0.180
```

With a bonus of 0.3 the shaped optimum commands *left* at cell 14, which leads to cell 13, 10 or 14 and never to the goal. Under the shaped reward its start value is 2.15, far above the 0.180 of the original optimum; under the original reward its value is zero, because it never reaches the goal. The shaped values show why. Cell 13 is worth more than cell 14 under the shaped reward, 1.71 against 1.52, because a move from 13 toward 14 collects the bonus, and from 14 the agent can always return to 13 to collect it again. This is the loop of Section 14.1 with values attached: shuttling between cells 13 and 14 earns the bonus on every lap, which is worth more than the reward and bonus that entering the goal collects once. With a bonus of 0.1 the loop is worth less than the goal, and the shaped optimum reaches the goal again, but its true value is 0.115 rather than 0.180, so it is a different and worse policy. One plausible bonus changes the optimal policy completely, and a smaller one still changes it.

14.2.5.2 Potential-Based Shaping

The bonus that Section 14.1 proposed instead is a difference of potentials,

$$\tilde{r}(s, a, s') = r(s, a) + \gamma\Phi(s') - \Phi(s), \tag{14.24}$$

for a bounded function Φ on the states. Around a loop its terms cancel up to the discount, and the following proposition shows that it changes the value of every policy by the same amount.

Proposition (potential-based shaping). *Let $0 \le \gamma < 1$, let the terminal states be absorbing, as in* `TabularMDP`*, and let $\tilde{V}^\pi$ and $\tilde{Q}^\pi$ be the values under the shaped reward. Then for every policy π, state s and action a,*

$$\tilde{V}^\pi(s) = V^\pi(s) - \Phi(s), \qquad \tilde{Q}^\pi(s, a) = Q^\pi(s, a) - \Phi(s). \tag{14.25}$$

In particular $\tilde{V}^* = V^* - \Phi$ and $\tilde{Q}^*(s, a) = Q^*(s, a) - \Phi(s)$, and the shaped and the original MDP have the same optimal policies (Ng et al. 1999).

Proof. Along any trajectory the shaping terms of the first T steps telescope:

$$\sum_{t=0}^{T-1} \gamma^t\big(\gamma\Phi(s_{t+1}) - \Phi(s_t)\big) = \gamma^T\Phi(s_T) - \Phi(s_0). \tag{14.26}$$

As $T \to \infty$ the first term vanishes, because Φ is bounded and $\gamma < 1$. The shaped return of every trajectory is therefore its original return minus $\Phi(s_0)$. Taking expectations under π from s gives the first identity, and from (s, a) the second. At each state the shift $\Phi(s)$ is

the same for every policy and every action, so the ordering of policies and the maximizing actions of $Q^*(s,\cdot)$ are unchanged. ■

Thus a potential changes every value, but every policy feels the change equally, and the comparison between policies, which is all that optimization uses, stays the same. Exercise 11c checks $\tilde{V}^* = V^* - \Phi$ on the lake. Ng et al. (1999) also proved a converse. If an added term $F(s,a,s')$ is not a difference of potentials, then there are a kernel and a reward on the same states and actions for which no optimal policy of the shaped MDP is optimal for the original one. The approach bonus is such a term, and the slippery lake is such an MDP.

If a potential cannot change the optimal policy, why use one? For exact planning there is no reason, since value iteration finds the same policy with or without it. The purpose is a learner that sees only sampled transitions. A potential that rises toward the goal, such as $\Phi = -0.3\,\ell$, spreads the goal's reward over the route, so that a learner receives a signal on every move instead of only on the last one.

The proof used one property of `TabularMDP`: its terminal states are absorbing, so trajectories continue forever and the term $\gamma^T \Phi(s_T)$ vanishes in the limit. A simulator stops at the first terminal state instead. If that happens after T steps, the shaped return of the episode is its original return plus $\gamma^T \Phi(s_T) - \Phi(s_0)$. The term $\gamma^T \Phi(s_T)$ depends on which terminal state the episode ends in, and when, so it can change which policy is best, unless Φ is zero at every terminal state. This is the condition that Section 14.1 stated in words. The potential $\Phi = -0.3\,\ell$ is zero at the goal but negative at the holes. In a simulator it would therefore charge the agent for falling in, more for an early fall than for a late one, a penalty the original task does not contain.

Everything in this section read the table P: every backup averaged over the next states with their exact probabilities. We can now compute the value of every state and every command of a known finite MDP, find an optimal policy with a certificate, and check what a change of reward does to it. A learning agent is rarely given the table. It sees one transition at a time, the one its own action produced. Section 14.3 asks whether values and an optimal policy can be learned from such samples alone.

14.2.6 Summary

Dynamic programming computes the best policy of a known MDP from the values of its states. The value $V^\pi(s)$ is the expected discounted return from s under π, and the Bellman equations tie each value to those of its successors: by an average over actions for a fixed policy, by a maximum for the optimal one. For $\gamma < 1$ the Bellman operator $\mathcal{T}$ is a γ-contraction, so value iteration converges geometrically to the unique V^* from any start and the change between sweeps certifies its error; policy iteration alternates exact evaluation, one linear solve, with a greedy improvement. On the slippery lake the optimal policy reaches the goal about sixteen times as often as the shortest path, because in every cell it chooses the command with the least risk of a hole, even when that command points away from the goal. A bonus that pays for approaching the goal can change the optimal policy, while a bonus that is a difference of potentials cannot.

14.2.7 Exercises

1. code **Policy evaluation as a linear system.** By Equation 14.23, $V^\pi = M_\pi r_\pi$ with $M_\pi = (I - \gamma P_\pi)^{-1}$.
   1. Show that $I - \gamma P_\pi$ is invertible for $0 \le \gamma < 1$ and that $M_\pi = \sum_{t \ge 0} \gamma^t P_\pi^t$.

   2. Interpret the entry $M_\pi(s, s')$ in terms of the visits to s' by an agent that starts at s and follows π in the terminating process of problem 4b of Section 14.1. Conclude that every row of M_π sums to $1/(1-\gamma)$, so that $(1-\gamma)M_\pi(s, \cdot)$ is a distribution over states, the *discounted state-visitation distribution* of π from s. Write $V^\pi(s)$ as $1/(1-\gamma)$ times the expectation of r_π under this distribution.
   3. Compute M_π for the policy `uniform` on the slippery lake and check that $M_\pi r_\pi$ reproduces the start-state value 0.0078 printed by `policy_evaluation`. Episodes of this policy last fewer than eight steps on average, yet the row of the start state sums to 20. Where does the rest of the mass sit?
2. **Stopping certificate.** The certificate Equation 14.21 bounds the unobservable error $\|V_k - V^*\|_\infty$ by the observable change $\|V_k - V_{k-1}\|_\infty$.
   1. Derive it from the contraction proposition of Section 14.2.2.3 by bounding the tail $\sum_{j \geq k} \|V_{j+1} - V_j\|_\infty$ of a telescoping sum. How small must the sweep-to-sweep change be to guarantee an error below a given tolerance?
   2. Derive an a priori bound that fixes the number of sweeps before value iteration starts and uses only γ, the tolerance and $\|V_1 - V_0\|_\infty$. Evaluate it for the slippery lake with $V_0 = 0$ and a tolerance of 10^{-6}, and compare it with the 164 sweeps that the certificate needed in the value iteration of Section 14.2.2 on this lake.
3. **The undiscounted lake.** The contraction proposition of Section 14.2.2.3 needs $\gamma < 1$, yet value iteration on FrozenLake also converges at $\gamma = 1$. In FrozenLake the holes and the goal are absorbing states with zero reward, and the only nonzero reward, 1, is collected on entering the goal.
   1. Repeat the proof of the proposition with $\gamma = 1$. What does it still show? Then show that whenever the operator $\mathcal{T}$ of Equation 14.18 has a fixed point at $\gamma = 1$, it has infinitely many.
   2. Show that value iteration from $V_0 = 0$ at $\gamma = 1$ produces a nondecreasing sequence $V_0 \leq V_1 \leq V_2 \leq \cdots$ that is bounded by one, so that it converges. What does $V_k(s)$ measure?
   3. Construct a two-state MDP with a single action on which V_k grows without bound at $\gamma = 1$. Which property of FrozenLake used in part b does it lack?
4. **Ties in policy iteration.** A common implementation of policy iteration stops when the greedy policy equals the current one, `(new == pi).all()`. On the slippery lake, *left* and *right* are both optimal at cell 6. On the 8×8 map, *left* and *up* are both optimal at state 51, and rounding errors can make the greedy policy of value iteration switch between them long after consecutive iterates differ only in the last digits.
   1. Explain how ties can keep this loop from terminating.
   2. The loop of Section 14.2.3.2 stops instead when the greedy step gains nothing at any state. Use the policy-improvement proposition to show that this test is reached after finitely many rounds, whichever way ties are broken, and that the policy at which it stops is optimal. Why does the test need a tolerance such as 10^{-12} in floating point?

   Adapted from Sutton and Barto, Reinforcement Learning: An Introduction, 2nd edition, Exercise 4.4.
5. code **Discount on calm and slippery ice.** On calm ice a cell j moves from the goal has optimal value γ^{j-1}, as Figure 14.6 shows.
   1. Run `value_iteration` on the calm map for $\gamma \in \{0, 0.5, 0.9, 0.99\}$ and compare each greedy policy with `shortest`. Use the closed form to explain why the policy does not depend on γ for $\gamma > 0$. What is special about $\gamma = 0$?
   2. Repeat on the slippery map for $\gamma \in \{0.5, 0.9, 0.99\}$. At which states does the optimal action change? For each of the three optimal policies, compute the proba-

bility of eventually reaching the goal. By the argument of problem 3b, this probability is the policy's value at $\gamma = 1$, which a few thousand sweeps of `policy_evaluation` on `d2l.TabularMDP.from_gym(env, 1.0)` compute.

3. Explain the trend in these probabilities. Why do slips make the optimal policy depend on γ?

6. code **In-place sweeps.** ⊠ `value_iteration` computes all of V_{k+1} from V_k. An *in-place* sweep instead visits the states in a fixed order and overwrites $V(s)$ immediately, so that states later in the same sweep already use the new values.
   1. Show that an in-place sweep is a γ-contraction in the sup norm for every order of the states, with the same fixed point as $\mathcal{T}$ of Equation 14.18. Conclude that the certificate Equation 14.21 remains valid for its iterates.
   2. Implement the in-place sweep and count the sweeps that the certificate needs to reach 10^{-6}, with the states in ascending and in descending order, on the slippery and the calm 4×4 and 8×8 maps. Compare the counts with those of `value_iteration`.
   3. Why does the order matter much more on calm ice than on slippery ice? Explain the calm-ice counts with Figure 14.6.

   Adapted from Sutton and Barto, Reinforcement Learning: An Introduction, 2nd edition, Sections 4.1 and 4.5.
7. code **Measured contraction rate.** ⊠ In Section 14.2.2 the error of value iteration shrinks by a factor of about 0.92 per sweep, faster than the guaranteed $\gamma = 0.95$. From sweep 14 on, the greedy policy of every iterate is `pi_star`.
   1. Show that from then on the error obeys the linear recursion $V_{k+1} - V^* = \gamma P_{\pi^*}(V_k - V^*)$, with P_{π^*} the kernel under π^* of Equation 14.23, and that the error is zero at the holes and the goal. Conclude that the asymptotic rate is the spectral radius ρ of γP_{π^*} restricted to the nonterminal states, and compute ρ.
   2. Compute ρ at $\gamma = 0.999$, using the optimal policy at that discount, and predict the number of sweeps that value iteration needs to reach an error of 10^{-6}. The guarantee $\gamma^k \|V_0 - V^*\|_\infty$ needs more than $13{,}000$ sweeps. Check the prediction with `value_iteration`. Which property of FrozenLake makes ρ smaller than γ?
   3. The rule $\|V_k - V_{k-1}\|_\infty \le 10^{-6}$ of Section 14.2.2 stops at sweep 128. There the true error was 11.7 times the sweep-to-sweep change, well below the certificate's factor $\gamma/(1-\gamma) = 19$. Show that an error that shrinks geometrically with ratio ρ gives the factor $\rho/(1-\rho)$. Why can the certificate Equation 14.21 not use ρ in place of γ?
8. code **Improvement, quantified.** ⊠ The policy-improvement proposition of Section 14.2.3.2 says that a greedy step does not lower the value. The following identity, a matrix form of the performance-difference lemma (Kakade and Langford 2002), measures the gain. For a policy π', let $\mathcal{T}^{\pi'}$ be the policy-evaluation operator of Equation 14.22 for π', and let $M_{\pi'}$ be the matrix of problem 1.
   1. Show that for any two policies π and π'

$$V^{\pi'} - V^{\pi} = M_{\pi'}(\mathcal{T}^{\pi'} V^{\pi} - V^{\pi}), \tag{14.27}$$

      and that $(\mathcal{T}^{\pi'} V^{\pi} - V^{\pi})(s) = \sum_a \pi'(a \mid s)\,[Q^{\pi}(s,a) - V^{\pi}(s)]$.
   2. Use the identity to reprove the policy-improvement proposition. Show that the improvement is strict at every state s with $Q^{\pi}(s,\pi'(s)) > V^{\pi}(s)$.
   3. Apply the identity with π^* in the role of π to show that $V^{\pi'} \le V^*$ for every stationary policy π', stochastic ones included.
   4. Check the identity numerically at the start state for π = `uniform` and $\pi' = \pi^*$.

9. code **When the greedy policy is optimal.** ⊠ In Section 14.2.3.2 the greedy policy of value iteration is optimal from sweep 14 on, while the certificate needs 164 sweeps. Let V be any table, let $e = \|V - V^*\|_\infty$, and let π be greedy with respect to V, so that $\pi(s)$ maximizes `mdp.backup(V)[s]`. Write $\mathcal{T}^\pi$ for the policy-evaluation operator of Equation 14.22.
    1. Show that $V^\pi(s) \geq V^*(s) - 2\gamma e/(1-\gamma)$ at every state (Singh and Yee 1994), by inserting $\mathcal{T}^{\pi^*}V$ and $\mathcal{T}^\pi V$ between $V^* = \mathcal{T}^{\pi^*}V^*$ and $V^\pi = \mathcal{T}^\pi V^\pi$. Evaluate the bound at sweep 14 on the slippery lake. What does it guarantee there?
    2. Let the *action gap* Δ be the smallest difference $V^*(s) - Q^*(s,a)$ over all states s and all actions a that are not optimal at s, with Q^* from Equation 14.17. Show that $2\gamma e < \Delta$ implies that π is optimal.
    3. Compute Δ for the slippery lake from `mdp.backup(V_star)`, treating actions within 10^{-9} of the maximum as optimal, and find the first sweep at which the condition of part b holds. Compare it with sweep 14. Why can the greedy policy be optimal long before the condition holds?
10. code **Time limits under the optimal policy.** `gym.make` wraps FrozenLake in a time limit: after 100 steps the episode ends with `truncated=True` (Section 14.1.2.3).
    1. Propagate the state distribution d_t^π of problem 2b of Section 14.1 under `pi_star` instead of the random policy. Compute the probabilities that an episode reaches the goal, falls into a hole, or is truncated, and check the first against the 0.730 computed in Section 14.2.4. Compare the truncated fraction with the one that d_{100}^π gives for the random policy.
    2. If the limit is part of the task, the best command can depend on the number of steps that remain (problem 5). Show that V_k, the k-th iterate of value iteration at $\gamma = 1$ from $V_0 = 0$, is the largest probability of reaching the goal within k steps, attained by commanding the greedy action of V_{k-1} when k steps remain. Compute it for $k = 100$. How much more often than π^* does this policy reach the goal within the limit?
11. code **Potential-based shaping.** ⊠ The proposition of Section 14.2.5 holds for every potential when the terminal states are absorbing, as in `TabularMDP`, but only for potentials that vanish at the terminal states when episodes stop there, as in the simulator.
    1. Add a constant c to every entry of `mdp.r`, the absorbing states included. Show that the discounted return of every trajectory changes by the same amount, and find that amount. Which constant potential Φ satisfies $\gamma\Phi(s') - \Phi(s) = c$?
    2. Now charge the constant only while the episode runs: add c to $r(s,a)$ at the non-terminal states alone. For $c \in \{-0.02, -0.01, 0.01, 0.05\}$, compute the optimal policy with `value_iteration`, and report its action at state 0, its value under the original reward with `mdp.policy_value`, and its probability of reaching the goal within 100 steps, computed as in Section 14.2.4. Explain the effect of each sign of c with the constant potential and the terminal condition.
    3. Replace the approach bonus of `shaped` by the potential $\Phi(s) = -0.3\,\ell(s)$, where ℓ is the Manhattan distance `dist`, and average the shaped reward over the next state, which gives at every state

$$\tilde{r}(s,a) = r(s,a) + \sum_{s'} P(s' \mid s,a)\,[\gamma\Phi(s') - \Phi(s)]. \tag{14.28}$$

    Check that the values $\tilde{V}$ that `value_iteration` returns satisfy $\tilde{V}(s) = V^*(s) - \Phi(s)$ at every state, and that the shaped optimum has the same value under the original reward as `pi_star`; at cell 6, where *left* and *right* tie, the two policies may

differ. Then drop the shaping term at the terminal states, as a simulator that stops there would, and recompute the value of the optimum under the original reward. Finally set Φ to zero at the terminal states and repeat.

Adapted from Sutton and Barto, Reinforcement Learning: An Introduction, Exercises 3.15 and 3.16.

12. [extended] **Backup budgets.** ▯▯ Policy iteration in Section 14.2.3.2 needed only two rounds, each with one exact linear solve, while value iteration needed 164 sweeps to certify its values. With many states a solve is too expensive, and the evaluation is done by sweeps instead. Measure cost in *backups*: one backup evaluates the bracket of Equation 14.19 for one state-action pair, so a value-iteration sweep costs $|\mathcal{S}||\mathcal{A}|$ backups and an evaluation sweep of a deterministic policy costs $|\mathcal{S}|$. *Modified policy iteration* (Puterman 1994) interpolates between the two algorithms. It starts from $U_0 = 0$. Round k performs one value-iteration sweep, which also yields the greedy policy π_k of U_k, followed by m sweeps of Equation 14.22 for π_k that start from the current table. With the operator $\mathcal{T}^{\pi_k}$ of Equation 14.22, a round reads $U_{k+1} = (\mathcal{T}^{\pi_k})^{m+1} U_k$. With $m = 0$ it is value iteration, and as $m \to \infty$ it becomes policy iteration.
    1. Let $V_k = \mathcal{T}^k U_0$ be the iterates of value iteration. Show that $\mathcal{T}U_0 \geq U_0$ implies $V_k \leq U_k \leq U_{k+1} \leq V^*$ for every k and every m, and check that the premise holds on FrozenLake. Conclude that after the same number of rounds, modified policy iteration is never further from V^* than value iteration.
    2. Implement modified policy iteration and plot the error $\|U - V^*\|_\infty$ of the current table U after every sweep against the cumulative number of backups for $m \in \{0, 1, 5, 20, 100\}$ on the slippery 4×4 and 8×8 maps at $\gamma = 0.95$.
    3. Explain the shape of the curves for large m. Why does the error stall between greedy steps?
    4. Repeat part b at $\gamma = 0.99$. How does the m that reaches an error of 10^{-6} with the fewest backups move as $\gamma \to 1$?

14.3 Temporal-Difference Learning

The previous section found the best way across the slippery lake by reading its transition table: for every cell and every command, it summed over the three places the agent might slide to. An agent dropped onto a real lake has no such table. It can only act, see where it lands, and collect whatever reward follows. Each step shows it one successor, chosen by the environment, never the list of possible successors with their probabilities. Can the agent still learn how good each command is, and from that the best policy, from such single draws?

The answer was found by Richard Sutton for predicting values in 1988 and by Chris Watkins for choosing actions in 1989 (Sutton 1988; Watkins 1989; Watkins and Dayan 1992). Treat the one successor the agent saw as a one-sample estimate of the sum it cannot compute, and move the table entry a small step toward that estimate. Over many visits the steps average the samples, and the average is the backup of dynamic programming. The resulting method, *Q-learning*, needs no model of the environment. With a neural network in place of the table, it becomes the deep Q-learning of the next chapter.

Learning from samples raises two questions. The samples are noisy, so the steps must shrink as visits accumulate; how fast? And an entry for a command the agent never tries is never updated, so the agent must keep trying commands that its table rates poorly. This section answers the first question and meets the second with the simplest rule, acting at

random at a small rate; Section 14.4 studies what that rule costs. The section ends by using the same transitions a second way, to estimate the transition table itself and plan in the estimate.

```
import gymnasium as gym
```

14.3.1 The Sampled Backup

We start from the equation that value iteration solved and ask which part of it needs the model.

14.3.1.1 From the Bellman Operator to One Sample

Value iteration in Section 14.2 repeated one operation, the Bellman optimality backup. Written for action values instead of state values, the Bellman operator $\mathcal{T}$ maps a table Q to the table

$$(\mathcal{T}Q)(s,a) = r(s,a) + \gamma \sum_{s' \in \mathcal{S}} P(s' \mid s,a) \max_{a' \in \mathcal{A}} Q(s',a') \quad \text{for all } s \in \mathcal{S},\ a \in \mathcal{A}. \tag{14.29}$$

Each entry of $\mathcal{T}Q$ is the expected immediate reward plus the discounted value of acting greedily on Q from wherever the agent lands. Its fixed point is Q^*, and repeated application converges to Q^* from any table, because $\mathcal{T}$ shrinks the largest difference between two tables by the factor γ, the contraction argument of Section 14.2 applied to action values.

The model enters the backup in one place, the average over the successor s'. An average can be estimated from samples. Suppose the agent stands at s, commands a, receives reward r and lands in s'. The number $r + \gamma \max_{a'} Q(s',a')$ is random. On the lake the reward, too, depends on where the agent lands. Averaged over the successor and the reward, the number equals $(\mathcal{T}Q)(s,a)$. One transition is therefore an unbiased one-sample estimate of one entry of the backup.

This is why model-free control learns Q rather than V. The state-value backup $V(s) \leftarrow \max_a [r(s,a) + \gamma \sum_{s'} P(s' \mid s,a) V(s')]$ puts the maximum outside the average over s'. One transition shows what one action did and says nothing about the brackets of the other actions, so it cannot estimate their maximum. In the Q form each action has its own entry, the average sits inside that entry, and the maximum acts only on numbers the table already holds. The same form makes the greedy policy $\operatorname{argmax}_a Q(s,a)$ a lookup in one row of the table, again without the model.

Q-learning moves the visited entry a fraction α of the way toward the sample, which is called its *target*:

$$Q(s,a) \leftarrow (1-\alpha)\, Q(s,a) + \alpha\big(r + \gamma \max_{a'} Q(s',a')\big). \tag{14.30}$$

The step size $\alpha \in (0,1]$ says how far to move: $\alpha = 1$ replaces the entry by the target, and a small α averages many targets. Because the target is built from the table's own estimate at the next state, the update is said to *bootstrap*. Rearranged, the change is proportional to a single number,

$$\delta = r + \gamma \max_{a'} Q(s', a') - Q(s, a), \qquad Q(s, a) \leftarrow Q(s, a) + \alpha\, \delta. \tag{14.31}$$

The number δ is the *temporal-difference error* (Sutton 1988): what the pair appears to be worth one step later, minus what the table said before the step. Panel (d) of Figure 14.5 draws the substitution. The exact backup sums over every successor; the sampled one follows the single observed transition and keeps the maximum at the next state.

Bootstrapping is a choice. *Monte Carlo* estimation waits until the episode ends and uses the realized return as the target for $Q(s_t, a_t)$. That target is unbiased for the value of the policy that generated the episode, but it carries the randomness of every later transition. A *temporal-difference* target looks one step ahead and trusts the table for the rest. It is biased while the table is inaccurate and usually far less variable (Sutton 1988; Sutton and Barto 2018). Q-learning adds the maximum to the one-step target, and with it estimates Q^* instead of the value of the data-collecting policy.

14.3.1.2 One Transition by Hand

Start from a table of zeros on the slippery lake, with $\gamma = 0.95$ and, for easy arithmetic, $\alpha = 0.5$. Every target is zero until the agent first reaches the goal, because rewards are zero elsewhere and every entry it could bootstrap from is zero. Suppose the first success comes from cell 14, where the command *right* slides into the goal, cell 15, with probability $1/3$. The goal is terminal, so the target is the reward alone, and

$$Q(14, \text{right}) \leftarrow 0.5 \cdot 0 + 0.5 \cdot 1 = 0.5.$$

Nothing else in the table changes. Later the agent stands on cell 13, commands *right*, and slides to 14. Its target is $0 + 0.95 \cdot \max_{a'} Q(14, a') = 0.95 \cdot 0.5 = 0.475$, so $Q(13, \text{right}) \leftarrow 0.5 \cdot 0.475 \approx 0.24$. Before the update at 14, the same transition from 13 would have taught the agent nothing. Value thus flows backward from the goal one cell per visit, along the transitions the agent happens to make. Value iteration moved it one cell per sweep, along all transitions at once.

The targets are noisy. If the agent next commands *right* on 14 and slips up to cell 10, whose entries are still zero, the target is 0 and $Q(14, \text{right}) \leftarrow 0.25$. The entry jumps between targets of 1 and 0, and later between values bootstrapped from cells 10 and 14. Only the average of these targets is the backup, and how fast the step should shrink to form that average is the subject of the step sizes below.

14.3.1.3 Least Squares and Double Sampling

Why should such noisy steps lead anywhere? Hold the table fixed and average the temporal-difference error over the successor and the reward:

$$\mathbb{E}[\delta \mid s, a] = (\mathcal{T}Q)(s, a) - Q(s, a). \tag{14.32}$$

The expected update, $Q(s, a) \leftarrow Q(s, a) + \alpha\big((\mathcal{T}Q)(s, a) - Q(s, a)\big)$, is a damped step of value iteration on that one entry. Q-learning is value iteration executed one sampled entry at a time, each step perturbed by noise of mean zero.

A reader who has fit regression models will object that the natural way to fit a table to targets is least squares. Given N recorded transitions (s_i, a_i, r_i, s'_i), we could minimize the squared temporal-difference error by gradient descent,

$$\ell(Q) = \frac{1}{N} \sum_{i=1}^{N} \Big(Q(s_i, a_i) - r_i - \gamma \max_{a'} Q(s'_i, a') \Big)^2 . \tag{14.33}$$

The gradient of one term has two parts: one through $Q(s_i, a_i)$, and one through the maximum inside the target. The update Equation 14.31 keeps the first and drops the second, holding the target fixed. It is therefore called a *semi-gradient* step, and it is not gradient descent on ℓ.

Dropping the second part is what makes Q-learning correct. Suppose the data contain each pair (s, a) with frequency $d(s, a)$, the visitation distribution of the data-collecting policy, and that there are infinitely many of them. Conditioning on (s, a) splits each expected squared term into the square of its mean and the variance of the target:

$$L(Q) = \mathbb{E}_{(s,a)\sim d}\Big[\big(Q(s,a) - (\mathcal{T}Q)(s,a)\big)^2 \Big] + \mathbb{E}_{(s,a)\sim d}\Big[\mathrm{Var}\big(r + \gamma \max_{a'} Q(s', a') \;\big|\; s, a\big) \Big]. \tag{14.34}$$

The first term is the squared Bellman residual and is zero at Q^*. The second is the variance of the target over the successor and the reward, and it depends on Q whenever transitions are random. Minimizing L trades a small residual against a less variable target, so its minimizer need not be Q^*. Three states show the direction of the distortion. From s, the only action leads with probability $1/2$ each to u_1, which pays 1 and terminates, or to u_2, which pays 0 and terminates; so $Q^*(u_1) = 1$, $Q^*(u_2) = 0$ and $Q^*(s) = \gamma/2$. The variance of the target at s is $\gamma^2 (Q(u_1) - Q(u_2))^2/4$, and the minimizer of L lowers it by pulling the two successor values toward each other, away from 1 and 0. Exercise 4 computes how far.

Residual-gradient methods descend the full gradient of the squared residual, and they meet the same difficulty (Baird 1995). An unbiased estimate of that gradient needs two independent successors of the same pair, one for the target and one for the derivative, and an agent living through one trajectory sees one. This is the *double-sampling problem*. With deterministic transitions and rewards the variance term vanishes and the two approaches agree. Holding the target fixed returns with networks as a second, slowly updated copy of the network that computes the targets, the target network of Section 15.4.

14.3.1.4 Step Sizes and Convergence

The worked transition showed one entry receiving a stream of noisy targets $y_1, y_2, \ldots$. The plainest way to combine them is to average. With the step size $\alpha_n = 1/n$ at the n-th visit of the pair, the update reads $Q_n = Q_{n-1} + (y_n - Q_{n-1})/n = ((n-1)Q_{n-1} + y_n)/n$, and by induction

$$Q_n = \frac{1}{n} \sum_{k=1}^{n} y_k ,$$

the sample mean of the targets, whatever the initial value. If the targets were independent draws with a fixed mean, the entry would converge to that mean by the law of large numbers. Robbins and Monro gave conditions under which a decreasing step size has this effect in general (Robbins and Monro 1951):

$$\sum_{n=1}^{\infty} \alpha_n = \infty, \qquad \sum_{n=1}^{\infty} \alpha_n^2 < \infty. \tag{14.35}$$

The first condition lets the steps travel any distance, so the initial value is forgotten. The second makes the accumulated noise finite. The sample mean $\alpha_n = 1/n$ satisfies both. A constant step size violates the second, and its entry keeps fluctuating around the mean with a variance of order α for small α.

Proposition (convergence of tabular Q-learning). *Let the state and action sets be finite, the rewards bounded and $\gamma < 1$. If every pair (s, a) is updated infinitely often and its step sizes satisfy the Robbins–Monro conditions, then Q converges to Q^* with probability one* (Watkins and Dayan 1992; Jaakkola et al. 1994; Tsitsiklis 1994).

Averaging alone does not prove this, because the targets are computed from the table being learned and have no fixed mean. Residual-gradient descent also has an expected step that vanishes at a fixed point, the wrong one. What singles out Q-learning is the contraction: whatever the current errors, the expected step shrinks the largest of them, and the step-size conditions make the noise around these steps average out.

On calm ice, where every command moves where it says, each pair has one successor and the target carries no transition noise. The update with a constant α is then a damped backup, $Q(s,a) \leftarrow (1-\alpha)Q(s,a) + \alpha(\mathcal{T}Q)(s,a)$, which applied to all entries contracts with modulus $(1-\alpha) + \alpha\gamma < 1$. A step as large as $\alpha = 0.9$ is then safe.

The conditions concern the limit, not a finite budget. The sample mean gives every target the same weight, including the early ones computed from a table that was still mostly zero, which on the lake are far too small. With $\alpha_n = 1/n$ the number of updates Q-learning needs can grow exponentially in $1/(1-\gamma)$, here 20; with $\alpha_n = 1/n^{\omega}$ for $1/2 < \omega < 1$ it grows only polynomially (Even-Dar and Mansour 2003). Below we test $\alpha_n = 1/(1+n)$, the sample mean with the zero initial value counted as one more target. Our default, $\alpha_n = 1/(1+0.1\,n)$, also satisfies both conditions, and its step approaches ten times that of the sample mean. It weights later targets more heavily, the k-th one in proportion to about k^9 (exercise 5), so the estimate follows the improving targets.

14.3.1.5 Terminal Masking

On an episodic task the target needs one correction. A terminal state, here a hole or the goal, has no future, so its continuation value is zero and the target contains only the immediate reward:

$$Q(s,a) \leftarrow Q(s,a) + \alpha\Big(r + \gamma\,(1 - \mathbf{1}(s' \text{ terminal}))\max_{a' \in \mathcal{A}} Q(s',a') - Q(s,a)\Big). \tag{14.36}$$

The mask uses `terminated`, not `truncated`. As Section 14.1 explained, a time limit stops the recording at a state from which the process would go on, so its continuation is not zero.

Treating truncation as termination would teach the table that the last cell of every long episode is worthless. In code the correction is one factor, `gamma * (1 - terminated) * Q[s_next].max()`.

14.3.2 Q-Learning on the Slippery Lake

We now run the update on the slippery lake of Section 14.1 with $\gamma = 0.95$ and grade the result against the exact solution of Section 14.2. The learner touches the environment only through `reset` and `step`. The model `mdp`, the optimal values `V_star`, the optimal action values `Q_star` (one backup of V^*) and the optimal policy `pi_star` serve only for grading.

```
gamma, num_episodes, num_seeds = 0.95, 4000, 5
env = gym.make('FrozenLake-v1', is_slippery=True)
mdp = d2l.TabularMDP.from_gym(env, gamma)
V_star = d2l.value_iteration(mdp, num_iters=1000)[-1]
Q_star = mdp.backup(V_star)
pi_star = Q_star.argmax(axis=1)
```

14.3.2.1 The Behavior Policy

Q-learning needs a rule for choosing actions while it learns, its *behavior policy* π_e. Acting greedily on the current table fails in two ways. On a table of zeros every row is a four-way tie, and `np.argmax` returns the first index, so a greedy agent would always command *left* and never find the goal. And an action whose entry is currently low is never tried, so its entry is never updated, and the underestimate is permanent. The simplest repair acts at random with a small probability ϵ:

$$\pi_e(a \mid s) = \begin{cases} 1 - \epsilon + \epsilon/|\mathcal{A}| & a = \text{argmax}_{a'} Q(s, a'), \\ \epsilon/|\mathcal{A}| & \text{otherwise.} \end{cases} \tag{14.37}$$

This is the *ϵ-greedy* policy. It explores with probability ϵ and otherwise takes the greedy action, breaking ties in the argmax uniformly at random. The function draws one uniform number to decide which, then picks a random action or a random member of the set of maximizers:

```
def epsilon_greedy(q, epsilon, rng):
    """Explore with probability epsilon, else act greedily on the values q."""
    if rng.random() < epsilon:
        return int(rng.integers(len(q)))
    # Ties are coin flips: np.argmax would always pick action 0 on a zero
    # table, and an agent that only ever moves left never finds the goal.
    return int(rng.choice(np.flatnonzero(q == q.max())))
```

A table that knows nothing should explore a lot, and a table that is nearly right should mostly exploit. We therefore let ϵ fall over training. `linear_schedule` turns a start value, an end value and a number of steps into a function of the step that interpolates linearly and then holds the end value:

```
def linear_schedule(start, end, num_steps):
    """step -> value, interpolated from start to end, then held at end."""
    return lambda step: end + (start - end) * max(0.0, 1.0 - step / num_steps)
```

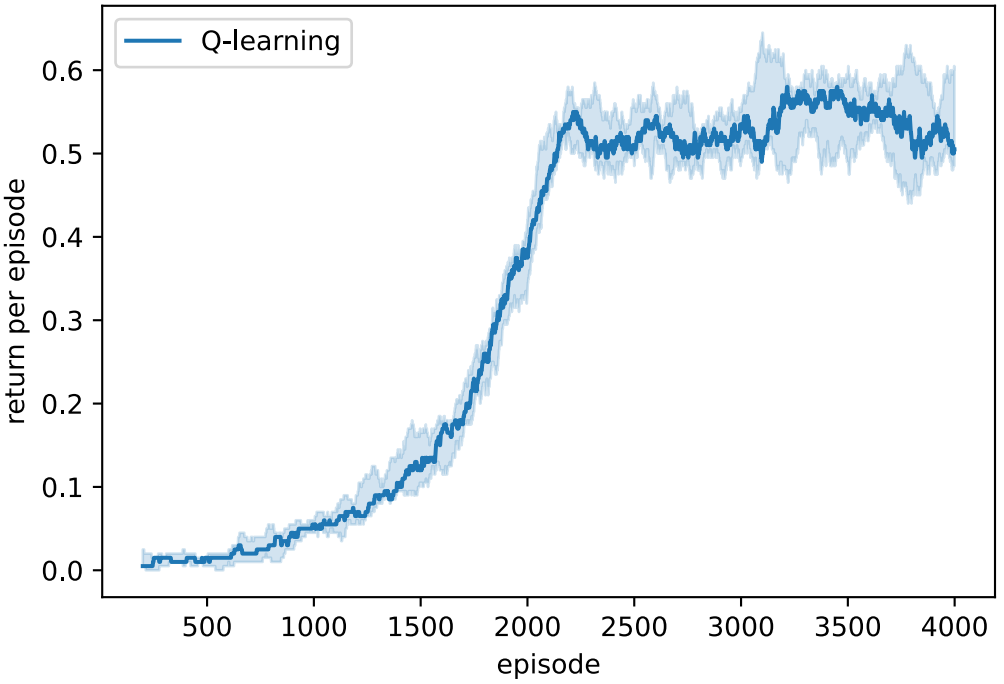
14.3.2.2 The Learning Loop

`q_learning` runs the update Equation 14.30 with the terminal mask. It takes the table `Q` and a table of visit counts `visits`, and changes both in place. The counts set each pair's step size through the function `alpha(n)`, and afterward they record which pairs were tried. The exploration rate is a function `epsilon(k)` of the episode, by default falling from 1 to 0.05 over the first half of training and held at 0.05 after that. The loop body chooses an action, observes one transition, and changes one entry by $\alpha\delta$; the function yields the return of each episode:

```
def q_learning(seed, Q, visits, env, num_episodes, gamma,
               alpha=lambda n: 1 / (1 + 0.1 * n), epsilon=None):
    """Tabular Q-learning; updates Q in place, yields each episode's return.
    `alpha(n)` is the step size at the n-th visit of a pair; `epsilon(k)` is
    the exploration rate in episode k (default: 1 to 0.05 over the first
    half of the episodes, then held)."""
    rng = np.random.default_rng(seed)
    if epsilon is None:
        epsilon = linear_schedule(1.0, 0.05, num_episodes // 2)
    env.reset(seed=seed)
    for episode in range(num_episodes):
        s, done, ret = env.reset()[0], False, 0.0
        while not done:
            a = epsilon_greedy(Q[s], epsilon(episode), rng)
            s_next, r, terminated, truncated, _ = env.step(a)
            visits[s, a] += 1
            delta = r + gamma * (1 - terminated) * Q[s_next].max() - Q[s, a]
            Q[s, a] += alpha(visits[s, a]) * delta
            s, done, ret = s_next, terminated or truncated, ret + r
        yield ret
```

Every cell in this section is plain NumPy, because a table needs only index arithmetic. We train one table per seed for $4{,}000$ episodes and plot the return of each episode:

```
Q = np.zeros((num_seeds, mdp.num_states, mdp.num_actions))
visits = np.zeros_like(Q)
returns = np.array([list(q_learning(k, Q[k], visits[k], env, num_episodes,
                                    gamma)) for k in range(num_seeds)])
d2l.plot_curves({'Q-learning': returns}, xlabel='episode',
                ylabel='return per episode', smooth=200)
```



Each point of the curve averages the 200 episodes that end there, so the curve starts at episode 200; the band spans the seeds. The return stays near zero while successes are rare, rises as value flows back from the goal, and flattens near 0.5 after the exploration rate reaches its floor at episode $2{,}000$.

14.3.2.3 The Learned Table and the Exact Solution

Is a return near 0.5 good? The optimal policy of Section 14.2 reaches the goal within the 100-step limit about 73% of the time. It maximizes the discounted return, which rewards speed, so a policy that ignores the discount succeeds about one point more often. The training curve, moreover, measures the behavior policy, which still acts at random one step in twenty, so it does not grade the table itself.

Because the model is known here, we can grade the table exactly. With $\gamma = 1$ and a reward of 1 only at the goal, the value of the start state is the probability of reaching the goal. Run for 100 sweeps from zero, `policy_evaluation` of Section 14.2 computes the probability of reaching it within 100 steps, and `success` wraps this for a policy given as one row of action probabilities per state. The mask `live` marks the cells where the agent acts, and `greedy` holds each table's greedy policy. The cell prints each table's value error and action-value error, the largest value a greedy action gives up under Q^*, the exact success rates, the success of π^* acting at random with probability 0.05, and the steps a run used:

```
lake1 = d2l.TabularMDP.from_gym(env, 1.0)    # gamma = 1: V(s0) = P(goal)
success = lambda pi: d2l.policy_evaluation(lake1, pi, 100)[-1][0]
live = ~np.isin(env.unwrapped.desc.flatten(), [b'H', b'G'])  # agent acts
greedy = np.eye(mdp.num_actions)[Q.argmax(axis=-1)]    # (seed, s, a)
optimal = np.eye(mdp.num_actions)[pi_star]
```

```
    taxed = 0.95 * optimal + 0.05 / mdp.num_actions        # pi* at epsilon = 0.05
    v_err = np.abs(Q.max(axis=-1) - V_star).max(axis=-1)   # one entry per seed
    q_err = np.abs(Q - Q_star)[:, live].max(axis=(1, 2))
    regret = (V_star - (greedy * Q_star).sum(-1)).max(-1)  # what greedy gives up
    wins = [success(g) for g in greedy]
    for name, per_seed in [('max_s |V_Qhat(s) - V*(s)|', v_err),
                           ('max_(s,a) |Qhat - Q*|', q_err),
                           ('max_s V*(s) - Q*(s, greedy)', regret),
                           ('greedy success in 100 steps', wins)]:
        print(f'{name:>27}: {np.round(per_seed, 3)}')
    steps = np.median(visits.sum(axis=(1, 2)))
    print(f'success of pi* {success(optimal):.3f}, of pi* at epsilon = 0.05 '
          f'{success(taxed):.3f}; median environment steps {steps:,.0f}')
```

```
  max_s |V_Qhat(s) - V*(s)|: [0.019 0.012 0.021 0.008 0.006]
      max_(s,a) |Qhat - Q*|: [0.131 0.079 0.113 0.087 0.075]
max_s V*(s) - Q*(s, greedy): [0. 0. 0. 0. 0.]
greedy success in 100 steps: [0.73 0.73 0.73 0.73 0.73]
success of pi* 0.730, of pi* at epsilon = 0.05 0.537; median environment steps 95,569
```

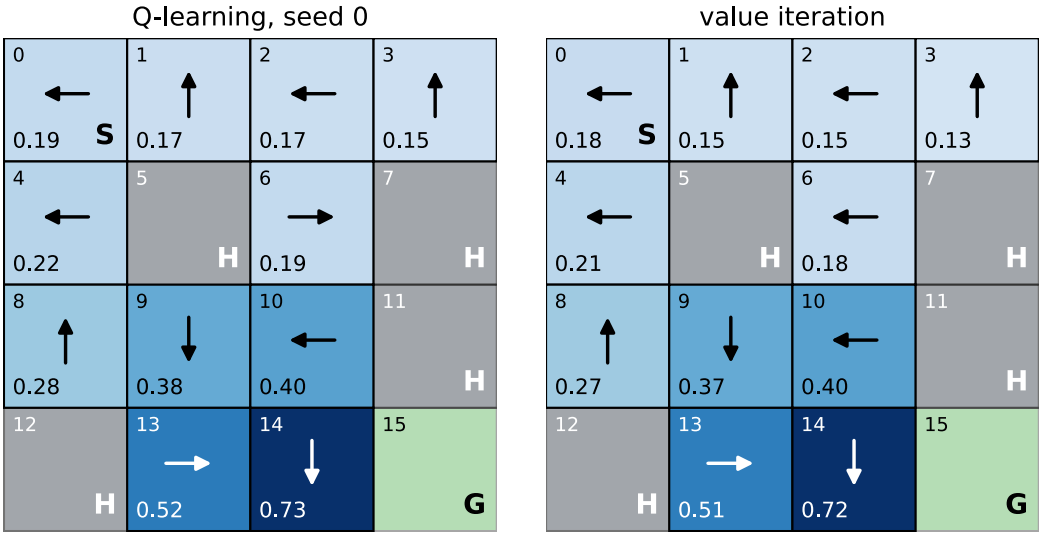
The third line is zero for every seed: in no cell does a learned greedy action lose value under Q^*, so each greedy policy is optimal up to ties and succeeds with the exact 73% of π^*. This is the typical run, not a guarantee at this budget. In about one run in eight some greedy action is still suboptimal after $4{,}000$ episodes, usually at little cost (success 71% to 74%). In the rare worst case the agent still prefers *right* at the start and succeeds less than half the time. The two best actions there differ by less than 0.01 in Q^*, the run's estimates were low, and a low estimate is corrected slowly, because the greedy rule tries that action only when it explores. The value error stays within about 0.03 of V^* across seeds. The action-value error is several times larger, about 0.05 to 0.17: the entries of actions the greedy policy does not take are updated mainly when the agent explores, so they average fewer targets.

Forced to act at random one step in twenty, π^* itself succeeds only about 54% of the time. The plateau of the training curve near 0.5 is this cost of exploration, not a defect of the table.

A run typically uses about $96{,}000$ environment steps. Value iteration certified V^* after 164 sweeps of 64 backups, about $10{,}500$ backups, but the two counts measure different things: a backup reads a whole row of the transition table, while a step reveals one sampled successor. Exercise 6 compares the two with these information requirements made explicit.

The learned values and arrows of the first seed, next to the exact solution:

```
d2l.show_grid(env.unwrapped.desc, np.stack([Q[0].max(-1), V_star]),
              np.stack([Q[0].argmax(-1), pi_star]),
              titles=['Q-learning, seed 0', 'value iteration'])
```



The learned arrows reproduce the slip-aware strategy of Section 14.2 except at cell 6, where Q-learning commands *right* and π^* commands *left*. Cell 6 lies between the holes 5 and 7, and the two commands have equal values under Q^*, so the disagreement is a different resolution of an exact tie. The values are most accurate in the cells the agent visits often and coarser in the rarely visited top-right corner.

14.3.2.4 Step-Size Schedules Compared

On calm ice the targets carry no transition noise, and the contraction argument above allows the constant step $\alpha = 0.9$. One eighth of the episode budget then almost always suffices, and here the greedy policy of the final table reaches the goal in every episode:

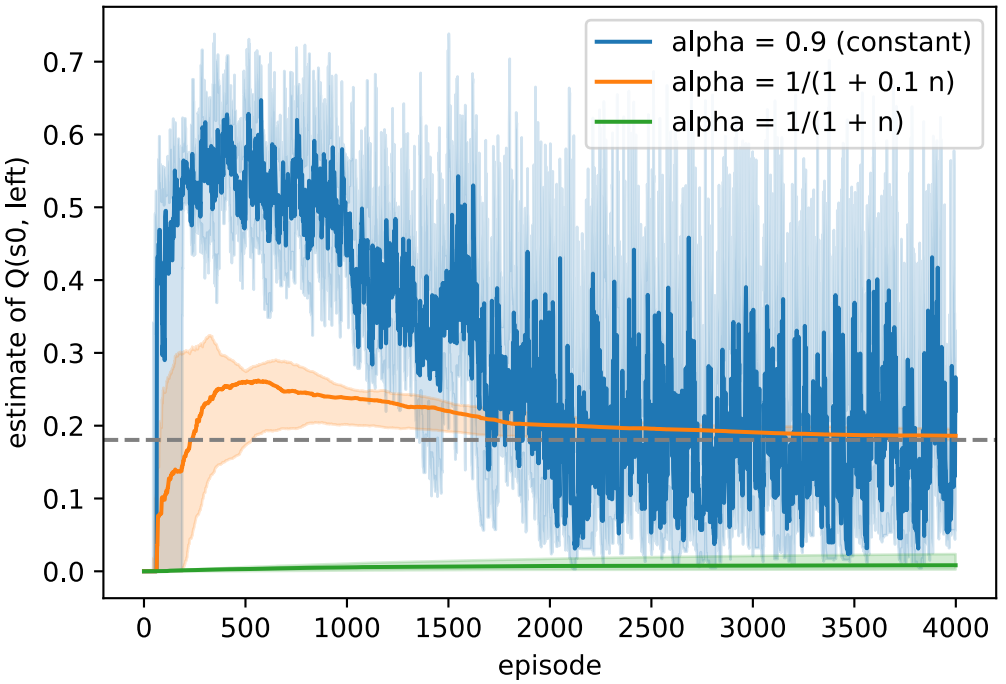
```
calm = gym.make('FrozenLake-v1', is_slippery=False)
table = np.zeros((mdp.num_states, mdp.num_actions))
calm_visits = np.zeros_like(table)
list(q_learning(0, table, calm_visits, calm, 512, gamma,  # fills table
                alpha=lambda n: 0.9))
print(f'calm ice, 512 episodes ({calm_visits.sum():,.0f} steps), alpha = 0.9: '
      f'greedy policy succeeds '
      f'{d2l.evaluate(calm, lambda s, rng: int(table[s].argmax()), 100):.0%}')
```

```
calm ice, 512 episodes (3,658 steps), alpha = 0.9: greedy policy succeeds 100%
```

The calm-ice run takes about $3{,}500$ environment steps, against about $96{,}000$ for a run on slippery ice. In the rare run that fails, the agent has reached the goal too seldom for its value to flow back to the start, an exploration failure that Section 14.4 studies. Transition noise, not the learning rule, sets most of the cost. On slippery ice the step size matters. We compare three schedules on the same seeds by tracking one entry, $Q(s_0, \text{left})$, whose exact value 0.180 is the value of the start under π^*. The plot shows the median over seeds with a band from the smallest to the largest:

```
a0 = int(pi_star[0])          # FrozenLake's actions: left, down, right, up
trace = {}
for name, alpha in [('0.9 (constant)', lambda n: 0.9),
                    ('1/(1 + 0.1 n)', lambda n: 1 / (1 + 0.1 * n)),
                    ('1/(1 + n)', lambda n: 1 / (1 + n))]:
    runs = []
    for seed in range(num_seeds):
        Qa = np.zeros_like(Q[0])  # updated in place: read after each episode
        runs.append([Qa[0, a0] for _ in q_learning(
            seed, Qa, np.zeros_like(Qa), env, num_episodes, gamma, alpha)])
    trace[f'alpha = {name}'] = np.array(runs)
    print(f'alpha = {name:>14}: final estimates {np.round(runs, 3)[:, -1]}')
print(f'peak of each 1/(1 + 0.1 n) run:  '
      f'{np.round(trace["alpha = 1/(1 + 0.1 n)"].max(axis=1), 3)}')
d2l.plot_curves(trace, xlabel='episode', ylabel='estimate of Q(s0, left)',
                reference=Q_star[0, a0])
```

```
alpha = 0.9 (constant): final estimates [0.056 0.221 0.206 0.265 0.236]
alpha =  1/(1 + 0.1 n): final estimates [0.19  0.186 0.195 0.184 0.18 ]
alpha =      1/(1 + n): final estimates [0.003 0.024 0.008 0.009 0.002]
peak of each 1/(1 + 0.1 n) run:  [0.228 0.291 0.262 0.324 0.205]
```



The constant step never settles. Each target moves the estimate by 90% of its error, so after $4{,}000$ episodes the final estimates scatter widely around 0.180, from near 0.05 to above 0.25. The default schedule ends within about 0.02 of the exact value in the typical run. The schedule $1/(1+n)$ satisfies the Robbins–Monro conditions as well, yet every run stays below 0.03, a sixth of the exact value, at this budget, because the early targets, computed from a table of zeros, keep their full weight. This is the slow rate that Even-Dar and Man-

sour predict. The conditions guarantee the limit; they do not choose a schedule for a finite budget.

14.3.2.5 Feedback between Estimation and Data Collection

The behavior policy depends on the current estimates, so errors in the estimates change which data are collected next. If $Q(s, a)$ is too large, the ϵ-greedy policy selects a more often and gathers transitions whose targets pull the estimate down. If $Q(s, a)$ is too small, the action is tried only when the agent explores, and, as the run above showed, the error is corrected slowly. Overestimates are thus self-correcting and underestimates are not, which is why the low run above took so long to change its first action.

At a fixed point of the update, the expected correction vanishes for every pair the behavior policy visits: $Q(s, a) = (\mathcal{T}Q)(s, a)$ there. This is not yet the optimality equation, because the maximum inside $\mathcal{T}Q$ still ranges over entries that are never visited and hence never constrained. A table can satisfy the equations on the visited pairs and still be wrong, which is why the convergence result needs every pair to be tried. An agent learning from a fixed log of transitions loses this feedback entirely; the next chapter returns to that case.

14.3.3 Off-Policy Learning

The maximum in the target makes Q-learning learn about one policy while following another. The two subsections below name that property and the bias that comes with it.

14.3.3.1 The Off-Policy Target and SARSA

The $\max_{a'}$ in Equation 14.31 evaluates a greedy continuation regardless of the action the behavior policy takes next at s'. Q-learning is therefore *off-policy*: it estimates the values of the greedy policy while the data come from the exploring policy π_e. The on-policy alternative bootstraps from the action a' that the behavior policy actually selects at s',

$$\delta = r + \gamma\, Q(s', a') - Q(s, a). \tag{14.38}$$

The method is called *SARSA*, after the tuple (s, a, r, s', a') that each update uses (Rummery and Niranjan 1994; Sutton and Barto 2018). For a fixed behavior policy, the expected SARSA update has the Bellman expectation equation of π_e as its fixed point, so SARSA learns Q^{π_e}, the values of the exploring policy with its random actions included. On the slippery lake, a SARSA agent that keeps $\epsilon = 0.05$ would value the cells next to holes below what π^* achieves there, because its own random steps sometimes fall in. Q-learning can explore persistently and still report a greedy policy that agrees with π^*, as it did above. The same property lets it learn from older experience, from another agent or from a fixed log, as long as the data cover the pairs the greedy policy needs. With networks in place of the table this flexibility can make learning unstable, as the next chapter shows.

14.3.3.2 Maximization Bias

The maximum of noisy estimates tends to exceed the maximum of their means, because the maximum picks out the entries whose errors happen to be positive. Since the maximum is convex, $\mathbb{E}[\max_{a'} \widehat{Q}(s', a')] \geq \max_{a'} \mathbb{E}[\widehat{Q}(s', a')]$, and every target that bootstraps from a noisy row inherits this *maximization bias*. The step-size traces above show it. Under

the default schedule, the estimate of $Q(s_0, \text{left})$ first climbs above 0.180 and only then settles. Without noise the estimate could not do this. From a table of zeros, with rewards that are never negative, every exact backup can only raise the values toward Q^*, so value iteration approaches 0.180 from below. The median trace instead sits at about 0.22 or above from episode 500 to episode $1{,}500$; noise, passed through the maximum, has pushed it up. The peaks of the printed runs lie between about 0.2 and 0.33, and over twelve seeds most runs overshoot in this way, by a few hundredths to more than a tenth. The constant step, whose estimates are noisier, overshoots further, to about 0.5 for much of the first thousand episodes.

The bias shrinks as the tabular estimates converge, but it can persist under function approximation, where the noise does not vanish. Double Q-learning keeps two estimators, uses one to select the maximizing action and the other to evaluate it, and so removes the upward bias, although its estimates can be low (Hasselt 2010). Exercise 7 builds it on a task where the bias misleads the agent.

14.3.4 Planning in a Learned Model

Q-learning uses each transition once. It moves one entry and discards the sample, and the next time the goal's value changes, the old transition is not there to carry the change backward. The same transitions can instead be counted. Let $N(s, a, s')$ count the observed transitions from s under a to s', and $N(s, a) = \sum_{s'} N(s, a, s')$. The counts give an estimated model,

$$\hat{P}(s' \mid s, a) = \frac{N(s, a, s')}{N(s, a)}, \qquad \hat{r}(s, a) = \frac{1}{N(s, a)} \sum_{\text{visits of } (s,a)} r,$$

the empirical frequency of each successor and the mean reward of each pair. Value iteration on $(\hat{P}, \hat{r})$, treating the estimate as if it were exact, is called *certainty equivalence*. It turns the problem without a model back into the problem of Section 14.2.

`Counts` wraps an environment: it passes every step through unchanged and records $N(s, a, s')$ and the summed rewards. Since `q_learning` touches the environment only through `reset` and `step`, it runs unchanged on the wrapper and learns the same table as it would without the wrapper. `learned_model` divides the counts into $\hat{P}$ and $\hat{r}$ and returns them as a `d2l.TabularMDP`. Rows of pairs that were never tried stay zero, so the planner treats them as worthless dead ends:

```
class Counts(gym.Wrapper):
    """Pass every step through and count it: N[s, a, s'] and summed rewards."""
    def __init__(self, env):
        super().__init__(env)
        n_s, n_a = env.observation_space.n, env.action_space.n
        self.N, self.R = np.zeros((n_s, n_a, n_s)), np.zeros((n_s, n_a))

    def reset(self, **kwargs):
        self.s, info = self.env.reset(**kwargs)
        return self.s, info

    def step(self, a):
```

```
        s_next, r, terminated, truncated, info = self.env.step(a)
        self.N[self.s, a, s_next] += 1
        self.R[self.s, a] += r
        self.s = s_next
        return s_next, r, terminated, truncated, info

def learned_model(counts, gamma):
    """Certainty equivalence: the empirical kernel and mean rewards."""
    n = np.maximum(counts.N.sum(axis=-1), 1)   # unvisited pairs: zero rows
    return d2l.TabularMDP(counts.N / n[..., None], counts.R / n, gamma)
```

How much better is the model's use of the data? We give Q-learning a short budget of 500 episodes and a longer one of $2{,}000$, each a full run with its own exploration schedule. On the transitions of each run we compare two tables: the one Q-learning learned, and the action values of the estimated model, which are one backup of its value-iteration solution. `grade` returns a table's sup-norm value error against `V_star` and the exact success of its greedy policy, computed by `success`:

```
def grade(Q, V_star, success):   # sup-norm error, success of greedy policy
    return (np.abs(Q.max(axis=-1) - V_star).max(),
            success(np.eye(Q.shape[-1])[Q.argmax(axis=-1)]))

for budget in [500, 2000]:
    rows = []
    for seed in range(num_seeds):
        counts, Qs = Counts(env), np.zeros((mdp.num_states, mdp.num_actions))
        list(q_learning(seed, Qs, np.zeros_like(Qs), counts, budget,
                        gamma))                    # fills Qs and the counts
        model = learned_model(counts, gamma)
        Q_model = model.backup(d2l.value_iteration(model, 500)[-1])
        rows.append([counts.N.sum(), *grade(Qs, V_star, success),
                     *grade(Q_model, V_star, success)])
    n, err_q, win_q, err_m, win_m = np.median(rows, axis=0)
    print(f'{budget} episodes, median {n:,.0f} steps '
          f'(medians over {num_seeds} seeds)')
    print(f'  Q-learning:        max |V - V*| {err_q:.3f}, success {win_q:.3f}')
    print(f'  planning in model: max |V - V*| {err_m:.3f}, success {win_m:.3f}')
```

```
500 episodes, median 9,984 steps (medians over 5 seeds)
  Q-learning:        max |V - V*| 0.084, success 0.486
  planning in model: max |V - V*| 0.096, success 0.639
2000 episodes, median 47,282 steps (medians over 5 seeds)
  Q-learning:        max |V - V*| 0.023, success 0.730
  planning in model: max |V - V*| 0.011, success 0.730
```

With little data, about $10{,}000$ steps, the planned greedy policy is the better one. It typically reaches the goal about two times in three, against about one in two for Q-learning's, and it was at least as good in about nine runs out of ten. The worst-case value error at this budget is set by a few rarely visited cells. It is usually smaller for the model, but not always, and the five runs printed here happen to favor Q-learning. With about $47{,}000$ steps both greedy policies are optimal in most runs, and the planned values are about twice as accurate. They were better in nearly every seed, and about as accurate as Q-learning's tables after twice

as many episodes. The reason is the one the worked transition showed. Q-learning moves value backward one transition per visit, using targets computed from a table that was wrong when the transition arrived. The model keeps every transition, and each of the few hundred sweeps of value iteration propagates the goal's reward through all of them. Exercise 1 shows the same effect in a two-state process.

The model has costs of its own. It stores $|\mathcal{S}|^2|\mathcal{A}|$ counts, here about a thousand numbers and for a large state space far too many, and every new batch of data means solving it again. The planner also trusts the estimate completely: a pair that was never tried looks worthless, and a pair that was tried once and got lucky looks better than it is. Section 15.6 fits such a model on Pendulum with regression in place of counts and plans in it.

The agent can now learn optimal action values from samples alone and act on them, or turn the same samples into a model and plan. Both rest on how it collects data, and the rule used here is crude. Forced to act at random one step in twenty, even π^* succeeds only about 54% of the time instead of 73%, and a greedy table needs a finite set of actions to maximize over. Section 14.4 asks whether a different rule can learn as much while losing less; later in the chapter the policy itself becomes the object that is learned.

14.3.5 Summary

Q-learning replaces the expectation in the Bellman backup by one observed transition and moves the visited entry a step α along the temporal-difference error δ. The expected step is a damped value-iteration step, and because the Bellman operator contracts, tabular Q-learning converges to Q^* when every pair is tried infinitely often and the step sizes satisfy the Robbins–Monro conditions. It is a semi-gradient method: gradient descent on the squared temporal-difference error converges elsewhere when transitions are random, the double-sampling problem. On the slippery lake its greedy tables are optimal in the typical run, the step-size schedule decides how fast it gets there, and the maximum in the target makes it off-policy and biases its estimates upward while they are noisy. Counting the same transitions and planning in the estimated model gives more accurate values from the same data, at the cost of storing and solving the model.

14.3.6 Exercises

1. **Batch TD against Monte Carlo.** A Markov reward process is an MDP without actions: each state has a fixed distribution over the reward and the successor state. One such process has two nonterminal states x and y, and $\gamma = 1$. A batch holds eight episodes: one moves from x to y with reward 0 and then terminates with reward 0; six start in y and terminate with reward 1; one starts in y and terminates with reward 0. TD(0) is the state-value form of Equation 14.31 without the maximum, $V(s) \leftarrow V(s) + \alpha\big(r + \gamma V(s') - V(s)\big)$.
    1. Compute the Monte Carlo estimates of $V(x)$ and $V(y)$ from the batch, and the values to which TD(0) converges when it replays the batch many times with a small step size.
    2. Which pair of estimates has the smaller squared error on the eight observed returns?
    3. Estimate the transition probabilities and the expected rewards of the process by counting, and solve its Bellman equation exactly. Which of the two pairs of estimates does the solution reproduce?

4. Which pair would you expect to predict the returns of future episodes from x better, and under which assumption about the process?

Adapted from Richard S. Sutton and Andrew G. Barto, Reinforcement Learning: An Introduction, Example 6.4.

2. **Truncation treated as termination.** A task has one state and one action, pays reward 1 at every step, and never terminates. A time limit truncates every episode after H steps, as `FrozenLake-v1` does after 100. A faulty implementation masks the bootstrap with `terminated or truncated` instead of `terminated`.

   1. With decaying step sizes, the estimate converges to the value at which the expected update over the H transitions of an episode vanishes. Find this value as a function of γ and H, and compare it with the correct value $1/(1-\gamma)$ at $\gamma = 0.95$ and $H = 100$. Which discount factor does the faulty implementation effectively use?
   2. Suppose instead that the time limit is part of the task, so that the return really ends after H steps. Show that no single number is the correct value of the state. Add the number h of remaining steps to the state, as in problem 5 of Section 14.1. Find the value of each state (s, h), and show that the mask `terminated or truncated` is then correct.

3. **Greedy SARSA and Q-learning.** SARSA replaces the maximum in Equation 14.31 by the value of the action a' that the behavior policy selects at s', as in Equation 14.38. An implementation selects a' before it updates $Q(s, a)$ and then takes a' at the next step. Suppose that both methods select actions greedily on their current tables and break ties by the same fixed rule. Are SARSA and Q-learning then the same algorithm, making the same action selections and the same updates? If they differ, construct a transition of the calm lake and table values for which the two methods select different actions at the next step.

   Adapted from Richard S. Sutton and Andrew G. Barto, Reinforcement Learning: An Introduction, Exercise 6.12.

4. code **The double-sampling counterexample.** ⊠ Build a three-state MDP with one action per state. From state s, the action gives reward 0 and moves to u_1 or u_2 with probability $1/2$ each. From u_1, it gives reward 1 and terminates; from u_2, it gives reward 0 and terminates. Write $Q(s)$, $Q(u_1)$ and $Q(u_2)$ for the three entries.

   1. Differentiate one term of the sampled objective Equation 14.33 with respect to the table. Which entries have nonzero derivatives, and which of these derivatives does the semi-gradient update Equation 14.31 drop?
   2. Write down Q^* for the three entries.
   3. With d uniform over the three pairs, write the population objective Equation 14.34 explicitly and minimize it in closed form as a function of γ. Which entries does the minimizer distort relative to Q^*, and in which direction?
   4. Visit the three pairs equally often and run, with a decaying step size, both the semi-gradient update and stochastic gradient descent on the full derivative of part (a). Confirm that each converges to the point predicted in parts (b) and (c).
   5. Suppose that the environment can be reset to s, so that every visit to s yields two independent successors. Use one of them in the target and the other in the derivative of part (a). Show that the resulting stochastic gradient is unbiased for the gradient of the squared Bellman residual, and confirm by simulation that it

converges to Q^*.

5. code **Step sizes under a finite budget.** The section compares a constant step size with schedules $\alpha = 1/(1 + c\,n)$ for $c = 0.1$ and $c = 1$, where n counts the visits to the entry, including the current one. For the analysis, treat the targets that one entry receives as independent draws with mean m and variance σ^2.
    1. For a constant α, find the variance of the estimate after many updates. Evaluate it at $\alpha = 0.9$ for $Q(s_0, \text{left})$, taking as targets $r + \gamma V^*(s')$ with s' drawn from `mdp.P[0, a0]` and `V_star` from the section, and compare the resulting standard deviation with the spread of the final constant-step estimates that the section prints around the exact value 0.180. Which assumption of the analysis fails for this entry, and where does the rest of the spread come from?
    2. For $c = 1$, write the estimate after n visits as a weighted sum of the initial value and the n targets. Which weight does the zero initialization receive?
    3. For general c, write the weight of the k-th target after n visits as a product, and compare the logarithm of the product with an integral to show that the weight grows like $k^{1/c-1}$ in k for large $k \le n$. How does $c = 0.1$ treat the targets that were computed from the early, inaccurate table?
    4. Sweep $c \in \{0.01, 0.03, 0.1, 0.3, 1\}$ on the slippery lake, five seeds each, and plot against c both the final error $|\widehat{Q}(s_0, \text{left}) - Q^*(s_0, \text{left})|$ and the sup-norm error $\|\widehat{Q} - Q^*\|_\infty$ over all entries. Explain the shape of the curves with parts (a) and (c). Do the two error measures select the same c?

6. code **Samples against backups.** Take `Q_star` as ground truth.
    1. Record $\|\widehat{Q} - Q^*\|_\infty$ after every episode of `q_learning` on the slippery lake and plot it against the number of environment steps, which the `visits` ledger counts. On the same axes, plot the error of value iteration, with Q_k = `mdp.backup` of the k-th iterate of `d2l.value_iteration`, against the number of Bellman backups at 64 per sweep. How many backups does value iteration need to reach the error that Q-learning reaches after 4000 episodes?
    2. Run `q_learning` on `Counts(env)`, and after 100, 400, 1000 and 4000 episodes solve `learned_model` with `d2l.value_iteration`, as in the section. Add the error of the model's action values to the plot at the same environment steps. How many environment steps does the model need to reach the error of Q-learning after 4000 episodes, and how many backups does each solve of the model cost?
    3. Value iteration consumes model backups, Q-learning consumes environment steps, and the estimated model of part (b) consumes both. Suppose that one environment step costs as much computation as b backups. Using your measurements, find for which range of b each of the three methods reaches the error of Q-learning after 4000 episodes at the lowest total cost. Which of the methods remain available to an agent that cannot read the transition kernel?

7. code **Maximization bias in a two-state task.** Every episode starts in state s_0, which offers two actions: *right* terminates with reward 0, and *left* moves to state s_1 with reward 0. In s_1, each of m actions terminates with a reward drawn from $\mathcal{N}(-0.1, 1)$, so *left* is a mistake. Use $\gamma = 1$, the constant step size $\alpha = 0.1$, zero initial values, and `epsilon_greedy` with $\epsilon = 0.1$.
    1. Implement tabular Q-learning for $m = 10$, and plot against the episode index the fraction of runs that take *left* in s_0, averaged over 1000 runs of 300 episodes.

What fraction would an ϵ-greedy agent acting on Q^* show?

2. Double Q-learning (Hasselt 2010) keeps two tables Q_1 and Q_2, acts ϵ-greedily on $Q_1 + Q_2$, and flips a fair coin at each transition. On heads it updates

$$Q_1(s,a) \leftarrow Q_1(s,a) + \alpha\big(r + \gamma\, Q_2(s', \operatorname{argmax}_{a'} Q_1(s',a')) - Q_1(s,a)\big), \tag{14.39}$$

and on tails it applies the same update with the two tables exchanged. Add its curve to the plot.

3. Every reward reachable through *left* has a negative mean. Explain how Q-learning's estimate of $Q(s_0, \text{left})$ nevertheless becomes positive. Predict how the Q-learning curve changes for $m = 1$ and for $m = 100$, and check. Which part of the effect at $m = 100$ comes from noisy estimates, and which from actions that have never been tried?

Adapted from Richard S. Sutton and Andrew G. Barto, Reinforcement Learning: An Introduction, Example 6.7.

8. code **SARSA and Q-learning on the cliff.** ☒ In `CliffWalking-v1` the agent crosses a 4×12 grid from the bottom-left corner to the bottom-right one. Every step costs -1, and a step into the cliff along the bottom edge costs -100 and returns the agent to the start. Use $\gamma = 1$, the constant step size $\alpha = 0.5$ and a constant $\epsilon = 0.1$.

   1. Write `sarsa`, a copy of `q_learning` that selects the next action a' with `d2l.epsilon_greedy` before the update, uses it in the target Equation 14.38, and then takes it. Train both methods for 500 episodes on ten seeds and plot the return per episode.
   2. Follow the greedy policy of each final table from the start. Which path does each method learn, and how long is it? Which method collects more reward while it learns, and why do the two answers differ?
   3. Let ϵ fall to zero over training. What happens to the two learned paths, and which values does each method learn in the limit?

Adapted from Richard S. Sutton and Andrew G. Barto, Reinforcement Learning: An Introduction, Example 6.6.

14.4 Exploration

Returning to a favorite restaurant gives a good meal; a new one may give a better meal or a worse one, and staying with the favorite forever means never finding out. The same question arises whenever we choose repeatedly among options whose value we learn only by choosing them: how often should we try something new, and when may we stop?

William Thompson posed the problem in 1933 for medicine: which of two treatments should the next patient receive, given how the earlier patients fared (Thompson 1933)? Herbert Robbins restated it in 1952 as a problem of sampling repeatedly from two populations, choosing each time which one to sample (Robbins 1952). It soon took the name *two-armed bandit*, after the slot machine, the one-armed bandit; with more than two options it is the *multi-armed bandit*. In 1985 Tze Leung Lai and Robbins showed that the cost of learning must grow with the number of trials, but that it can be kept as small as the logarithm of that number (Lai and Robbins 1985).

Q-learning in Section 14.3 faced the same question and answered it crudely. It learned only the entries of its table that it tried, and to keep trying commands that its table rated poorly,

it acted at random with probability ϵ. The rule has a visible cost: forced to act at random one step in twenty, the optimal policy on the slippery lake succeeds about 54% of the time instead of 73%. Is that the right amount of trying? Could a different rule learn as much while losing less?

The question matters again for language models trained with a verifier. The model samples several answers to a prompt, a checker scores them, and the model moves toward the answers that scored better. If every answer scores the same, there is nothing to move toward, and the model's own randomness is the only way it can find a better answer.

We first study the trade-off in a problem with a single state, the bandit, where its cost can be measured exactly. We then return to MDPs, where trying an untried pair can require a long sequence of actions, and end with the verifier task of Section 14.1.

```
import gymnasium as gym
```

14.4.1 The Multi-Armed Bandit

A bandit is an MDP with a single state. Each round the agent pulls one of several arms and receives a random reward. Nothing it does changes what comes next, and no reward is delayed, so only one question remains: which arm is best? The agent can answer it only by pulling arms, and every pull of a worse arm is a pull not spent on the best one.

14.4.1.1 Arms and Regret

We use ten arms with Bernoulli rewards: arm a returns 1 with probability μ_a and 0 otherwise. The best arm, the third, has $\mu^* = \max_a \mu_a = 0.90$; the second best has 0.60. The agent does not know the μ_a. After $n(a)$ pulls of arm a it knows the empirical mean $\hat{\mu}(a)$, the fraction of those pulls that returned 1. A *bandit rule* maps these tallies to the next arm. The function `bandit_run` plays one rule for `num_pulls` rounds, drawing all randomness from the generator `rng`; it keeps the tallies `wins` and `count` for every arm and passes them to the rule:

```
arms = np.array([0.50, 0.42, 0.90, 0.25, 0.55, 0.38, 0.60, 0.32, 0.50, 0.45])

def bandit_run(arms, rule, num_pulls, rng):
    """One state, one step: pull an arm, log the shortfall, update tallies.
    Returns the cumulative regret after each pull."""
    wins, count = np.zeros(len(arms)), np.zeros(len(arms))
    gap = np.empty(num_pulls)
    for t in range(num_pulls):
        a = rule(wins, count, t, rng)
        count[a] += 1
        wins[a] += float(rng.random() < arms[a])
        gap[t] = arms.max() - arms[a]
    return np.cumsum(gap)
```

The function records the shortfall of each pull rather than its reward. A success rate says how good a policy is; for a learner we want to know how much the learning cost. The pull of arm a_u in round u falls short of the best arm by $\mu^* - \mu_{a_u}$ in expectation, and the *regret* after t pulls adds up these gaps:

$$\text{Regret}(t) = \sum_{u=1}^{t} (\mu^* - \mu_{a_u}) = \sum_a \Delta_a \, n_t(a), \qquad \Delta_a = \mu^* - \mu_a. \tag{14.40}$$

The second form groups the pulls by arm: each arm's gap Δ_a times the number $n_t(a)$ of its pulls. Regret is what the agent loses against one that knew the best arm from the start. A rule that pulls one arm with mean 0.50 for good loses 0.40 per pull, 800 after 2000 pulls, and its regret grows linearly in t. If a rule's regret grows more slowly than t, the rule pulls worse arms ever more rarely, and its average regret per pull tends to zero.

14.4.1.2 Greedy and Epsilon-Greedy Rules

The rule of Section 14.3 works here unchanged. `d2l.epsilon_greedy(q, epsilon, rng)` takes a row of value estimates, here the ten empirical means. With probability ϵ it picks an arm uniformly at random; otherwise it picks the arm with the largest estimate, breaking ties at random. An arm that was never pulled has empirical mean 0. With $\epsilon = 0$ the rule is *greedy*. A fixed ϵ keeps exploring at the same rate forever, while an *annealed* rule lowers ϵ over time. We use $\epsilon_t = \min(1, 20/t)$ in round t, which pulls at random for the first twenty rounds and then ever more rarely. The schedule depends only on the round, not on anything the agent has learned.

Each of these rules turns a row of value estimates into a distribution over actions, as Figure 14.9 shows for a row of four action values, one per move of the lake. Greedy puts all probability on the largest estimate, and ϵ-greedy spreads a fraction ϵ of it uniformly over all actions. A third rule, *Boltzmann exploration*, makes the probability of each action proportional to $\exp(\hat{Q}(s,a)/\beta)$. This is the softmax of Section 3.1 with temperature β: a large β approaches the uniform distribution, a small one the greedy rule. Unlike ϵ-greedy, it tries an action that is nearly the best more often than one that is clearly bad.

The function `eps_rule` turns a schedule $t \mapsto \epsilon_t$ into a bandit rule that applies `d2l.epsilon_greedy` to the empirical means. The function `regret_runs` plays a rule in twenty independent runs and returns their regret curves; each run seeds its own generator from the stream number and the run index, so that different rules get independent runs. The function `report` prints the median and the quartiles of the final regret over the runs. We report medians because a few runs can dominate a mean:

```
def eps_rule(schedule):
    """Epsilon-greedy on the empirical means, with epsilon = schedule(t)."""
    return lambda wins, count, t, rng: d2l.epsilon_greedy(
        wins / np.maximum(count, 1), schedule(t), rng)

def regret_runs(arms, rule, stream, num_runs=20, num_pulls=2000):
    """Regret curves of `num_runs` independent runs, shape [run, pull]."""
    return np.array([bandit_run(arms, rule, num_pulls,
                                np.random.default_rng([2026, stream, s]))
                     for s in range(num_runs)])

def report(name, r):
    q25, q50, q75 = np.percentile(r[:, -1], [25, 50, 75])
    print(f'{name:>16}: regret after {r.shape[1]} pulls: median {q50:3.0f}'
          f' (quartiles {q25:.0f} to {q75:.0f})')
```

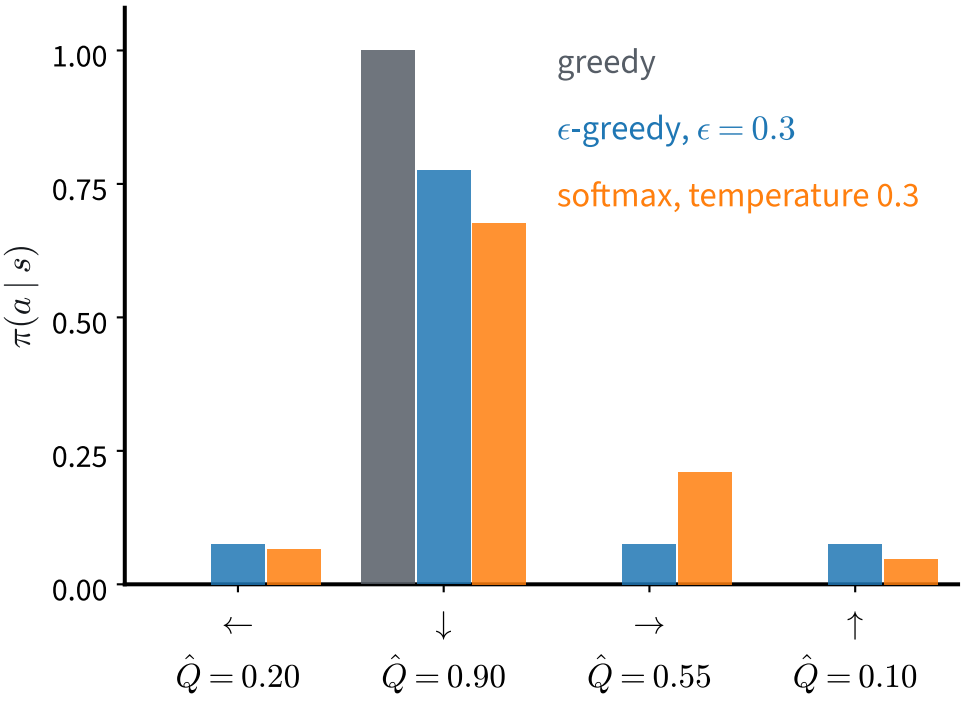


Figure 14.9: Three exploration rules applied to one row of action values, $\hat{Q}(s, \cdot) = (0.20, 0.90, 0.55, 0.10)$ for the four moves of the lake. Greedy puts all probability on the largest value. ϵ-greedy with $\epsilon = 0.3$ gives each action 0.075 and the rest to the largest value. The softmax with temperature 0.3 ranks the actions by their values.

```
rules = {'greedy': eps_rule(lambda t: 0.0),
         'epsilon = 0.1': eps_rule(lambda t: 0.1),
         'epsilon = 20/t': eps_rule(lambda t: min(1.0, 20 / (t + 1)))}
regret = {name: regret_runs(arms, rule, i)
          for i, (name, rule) in enumerate(rules.items())}
for name, r in regret.items():
    report(name, r)
```

```
       greedy: regret after 2000 pulls: median 800 (quartiles 700 to 980)
 epsilon = 0.1: regret after 2000 pulls: median 106 (quartiles 92 to 132)
epsilon = 20/t: regret after 2000 pulls: median  46 (quartiles 44 to 53)
```

The greedy rule does worst, and the reason can be worked out exactly. All empirical means start at 0, so the first pull goes to a random arm. If it returns 0, that arm's mean stays 0 and all ten arms are still tied. If it returns 1, its mean becomes positive and stays positive forever: one success keeps the fraction of wins above zero, while every other arm still reads 0. Greedy therefore pulls random arms until one returns a reward and then never pulls anything else. In each of those random pulls, arm a is chosen with probability $1/10$ and returns 1 with probability μ_a, so the arm it settles on is a with probability $\mu_a / \sum_b \mu_b$. The best arm wins this lottery with probability $0.90/4.87$, in about one run in five. A run that settles on arm a loses Δ_a on almost every pull, about $2000\,\Delta_a$ in all, so the expected regret is

$$\mathbb{E}\big[\text{Regret}(2000)\big] \approx 2000 \sum_a \frac{\mu_a}{\sum_b \mu_b}\,\Delta_a \approx 707.$$

A single run loses one of the values $2000\,\Delta_a$, from 0 (about one run in five) to 1300; the median, around 800, belongs to the runs that settle on an arm with mean 0.50.

The fixed rule with $\epsilon = 0.1$ finds the best arm in almost every run but never stops exploring. A random pull falls short of the best arm by the average gap, $0.90 - 0.487 = 0.41$, where 0.487 is the mean of the ten μ_a. Exploration alone therefore costs $0.1 \times 0.41 \approx 0.04$ per pull, about 83 of its median regret of about 100, and this cost does not shrink: the regret of a fixed ϵ grows linearly in t.

The annealed rule makes about 112 random pulls in 2000 rounds: twenty at the start and about $20 \ln(2000/20) \approx 92$ afterwards. At 0.41 each, they cost 46, close to its median regret of about 50. Nearly all of its regret is spent on exploration, and the greedy pulls in between go to the best arm. Since the number of random pulls up to round t grows like $20 \ln t$, so does the regret: an ϵ_t proportional to $1/t$ gives logarithmic regret. The constant matters. If it is too small, the rule stops exploring before it has told the best arm from the second best. The analysis of Auer et al. (2002) guarantees logarithmic regret on every bandit with m arms whose smallest gap is $\Delta_{\min}$, with $\epsilon_t = \min(1, c\,m/(\Delta_{\min}^2 t))$ and any $c > 5$. For these arms, with $\Delta_{\min} = 0.30$, that is nearly thirty times our $20/t$. Our constant was chosen for these arms, and the guarantee does not cover it.

14.4.2 Optimism and Posterior Sampling

The ϵ-greedy rules explore blindly. A random pull is as likely to go to the worst arm as to the second best, and an arm pulled a hundred times is as likely to be tried again as one pulled

twice. A better rule directs exploration to the arms whose means are still uncertain and might be the best, because only those pulls can change the decision.

14.4.2.1 Upper Confidence Bounds

One way to do this is optimism: treat each arm as if it were as good as the data still allow. An untried arm could be anything, so it gets tried. An arm pulled often has a precise estimate and is pulled again only if that estimate is high. The *upper confidence bound* (UCB) rule adds to each empirical mean a bonus that measures its uncertainty, and pulls the arm with the largest sum (Auer et al. 2002):

$$a_t = \operatorname*{argmax}_a \left[\hat{\mu}(a) + \kappa \sqrt{\log t \,/\, n(a)} \right]. \tag{14.41}$$

The bonus shrinks like $1/\sqrt{n(a)}$, the rate at which the error of an average of $n(a)$ rewards shrinks, and it grows slowly with t while arm a is not pulled, so no arm is given up for good. An arm is pulled either because its mean is high or because its mean is uncertain. Every arm is pulled once before the rule starts, since the bonus is undefined at $n(a) = 0$. Figure 14.10 shows the bonus and the indices of one run.

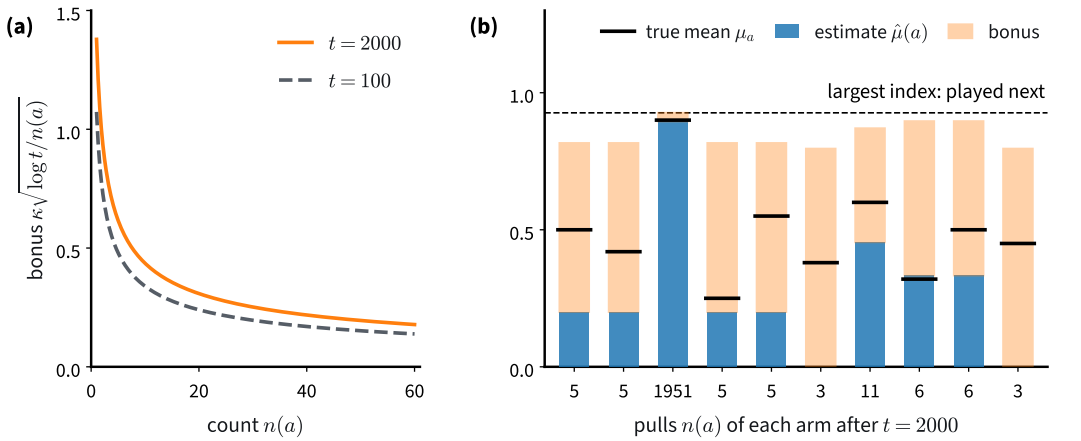


Figure 14.10: Optimism under uncertainty, with $\kappa = 0.5$. (a) The bonus $\kappa\sqrt{\log t/n(a)}$ shrinks with an arm's count $n(a)$ and grows slowly with time t. (b) One run of the UCB rule on the ten arms after 2,000 pulls: each bar is an arm's estimate $\hat{\mu}(a)$ with its bonus on top, labeled by the arm's pull count, and the black marks are the true means. Nearly all pulls went to the best arm, whose index is the largest; the rarely tried arms keep indices close to it, so each will be tried again as $\log t$ grows.

The size of the bonus comes from Hoeffding's inequality (Hoeffding 1963): the mean $\hat{\mu}_n$ of n independent rewards in $[0, 1]$ exceeds the true mean μ by more than h with probability at most e^{-2nh^2}, and likewise falls below it. With $h = \kappa\sqrt{\log t/n}$, the true mean lies above the index with probability at most $t^{-2\kappa^2}$, for a fixed count n. A larger κ makes such failures rarer and spends more pulls on bad arms. With $\kappa = \sqrt{2}$ the rule is UCB1 of Auer et al. (2002), whose regret grows logarithmically in t on every bandit with rewards in $[0, 1]$; the

same holds for every $\kappa > 1$ (Bubeck and Cesa-Bianchi 2012), and exercise 2 derives the bound.

14.4.2.2 Thompson Sampling

Thompson's rule of 1933 replaces the bonus by a belief. It keeps, for each arm, a posterior distribution over the arm's mean, draws one sample from each posterior, and pulls the arm with the largest draw. For rewards in $\{0, 1\}$ and a uniform prior on $[0, 1]$, the posterior after w wins in n pulls is the Beta distribution $\text{Beta}(1 + w, 1 + n - w)$. An arm with 3 wins in 4 pulls has the posterior $\text{Beta}(4, 2)$, with mean $2/3$ but wide. An untried arm has the uniform $\text{Beta}(1, 1)$, and its draw beats a draw from $\text{Beta}(4, 2)$ with probability $1/3$. As pulls accumulate, the posteriors narrow, and each arm is pulled with the posterior probability that it is the best one. Thompson sampling also has logarithmic regret (Agrawal and Goyal 2012), and it has no constant to choose.

14.4.2.3 UCB and Thompson Sampling Compared

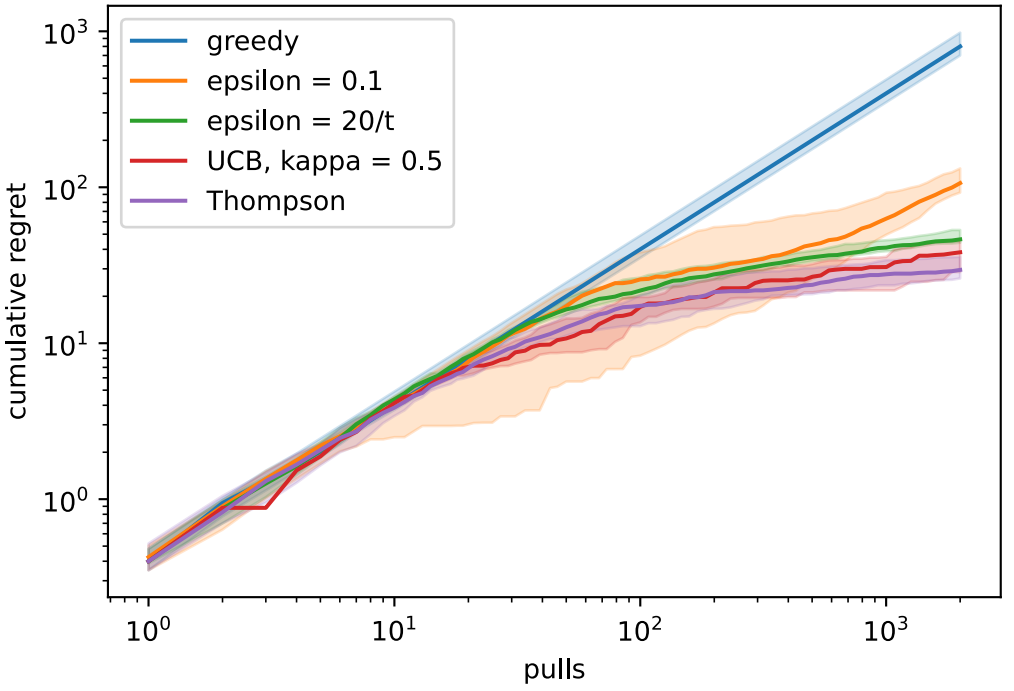
The function `ucb(kappa)` returns the UCB rule with coefficient κ, and `thompson` draws from the Beta posteriors. We run UCB with $\kappa = 0.5$, a value chosen for these arms that we examine below, and Thompson sampling, and then plot all five rules. `d2l.plot_curves` draws the median over the runs with the band between the quartiles, here at 100 pulls spaced evenly on logarithmic axes:

```
def ucb(kappa):
    """The UCB rule with coefficient kappa: each arm once, then the largest
    index mean + kappa * sqrt(log t / n)."""
    def rule(wins, count, t, rng):
        if (count == 0).any():          # play each arm once before pricing any
            return int(np.argmax(count == 0))
        bonus = kappa * np.sqrt(np.log(t + 1) / count)  # t + 1: round number
        return int(np.argmax(wins / count + bonus))
    return rule

def thompson(wins, count, t, rng):
    """Draw each arm's mean from its Beta posterior; play the largest draw."""
    return int(rng.beta(1 + wins, 1 + count - wins).argmax())

for i, (name, rule) in enumerate([('UCB, kappa = 0.5', ucb(0.5)),
                                  ('Thompson', thompson)], 3):
    regret[name] = regret_runs(arms, rule, i)
    report(name, regret[name])
pulls = np.unique(np.geomspace(1, 2000, 100).astype(int))  # even on a log axis
d2l.plot_curves({name: r[:, pulls - 1] for name, r in regret.items()},
                xlabel='pulls', ylabel='cumulative regret', x=pulls,
                xscale='log', yscale='log', band='iqr')
```

```
UCB, kappa = 0.5: regret after 2000 pulls: median  38 (quartiles 29 to 46)
       Thompson: regret after 2000 pulls: median  30 (quartiles 26 to 36)
```



Both rules end with a median regret of roughly 30 to 40, about a third of the fixed ϵ and below the annealed rule. They are close to each other: the difference between them is smaller than the spread over runs. On logarithmic axes a regret that grows linearly is a line of slope one. Greedy and fixed ϵ approach that slope, while UCB, Thompson sampling and the annealed rule bend away from it. 2000 pulls cannot establish an asymptotic rate, but the bend is what logarithmic growth looks like.

14.4.2.4 The UCB Coefficient

The value $\kappa = 0.5$ lies below the range $\kappa > 1$ that the theory covers. How much does the coefficient matter? The cell plays forty runs for each of three coefficients and prints the median and extremes of the regret:

```
for j, kappa in enumerate([0.15, 0.5, np.sqrt(2)]):
    r = regret_runs(arms, ucb(kappa), 10 + j, num_runs=40)[:, -1]
    print(f'kappa = {kappa:.2f}: regret after 2000 pulls over 40 runs: median '
          f'{np.median(r):3.0f}, min {r.min():3.0f}, max {r.max():3.0f}')
```

```
kappa = 0.15: regret after 2000 pulls over 40 runs: median   8, min   4, max 709
kappa = 0.50: regret after 2000 pulls over 40 runs: median  38, min  21, max  59
kappa = 1.41: regret after 2000 pulls over 40 runs: median 204, min 173, max 241
```

At 2000 pulls the coefficient decides the result. UCB1, $\kappa = \sqrt{2}$, ends near 200, five to six times the regret of $\kappa = 0.5$. With $\kappa = 0.15$ the median is the smallest, about ten, but the maximum is large. With so small a bonus, a run in which the best arm starts with a few unlucky pulls can settle on a worse arm and keep losing 0.3 or more per pull. This happens in roughly one run in ten; the median hides these runs, and the mean does not.

One might expect the theory's κ to win once the horizon is long enough. On these arms it does not: in runs of 10^5 pulls and more, which exercise 4 reproduces, $\kappa = \sqrt{2}$ keeps six to eight times the regret of $\kappa = 0.5$, and both grow like $\log t$. The bonus accounts for the factor. UCB keeps pulling a worse arm a until its bonus $\kappa\sqrt{\log t / n(a)}$ has shrunk to about Δ_a, that is for about $\kappa^2 \log t / \Delta_a^2$ pulls, so in long runs the ratio of the two regrets approaches $(\sqrt{2}/0.5)^2 = 8$. What the theory guarantees is something else: $\kappa = \sqrt{2}$ has logarithmic regret on every bandit with rewards in $[0, 1]$, and $\kappa = 0.5$ has no such guarantee. Hoeffding's inequality must hold for the worst case, an arm with variance $1/4$. The best arm here has variance $0.9 \times 0.1 = 0.09$, so its empirical mean is more accurate than the inequality assumes, and a smaller bonus is enough on this problem. Thompson sampling matches the tuned UCB rule without a constant to tune.

14.4.2.5 A Lower Bound on Regret

How low can regret go? Lai and Robbins proved that any rule that does well on every bandit of a family must, in t pulls, pull each worse arm a at least about $\ln t / D(\mu_a \,\|\, \mu^*)$ times (Lai and Robbins 1985). Here

$$D(p \,\|\, q) = p \ln \frac{p}{q} + (1 - p) \ln \frac{1 - p}{1 - q}$$

is the relative entropy between Bernoulli distributions with means p and q. It is small when the two means are close, so an arm that is nearly as good as the best is hard to tell apart and must be pulled more often. For the arm with mean 0.60, $D(0.60 \,\|\, 0.90) = 0.31$, and it must be pulled about $\ln 2000 / 0.31 \approx 24$ times by round 2000. With Equation 14.40, the regret of every such rule grows at least like $\ln t$. UCB and Thompson sampling grow at this rate, so in long runs no rule does better by more than a constant factor; fixed ϵ, whose regret grows linearly, does not reach this rate.

14.4.3 Exploration in MDPs

In a bandit every arm can be pulled in every round. In an MDP, a state-action pair can be tried only from its state, and reaching that state may take a particular sequence of actions. Random actions produce a given sequence with a probability that falls geometrically in its length: with two actions, a particular sequence of k actions comes up with probability 2^{-k}.

14.4.3.1 A Chain of States

A chain makes this exact. It has N states in a row and starts in state 0. Action 1 moves one state on, action 0 returns to the start, and entering the last state gives reward 1 and ends the episode, which is cut after $2N$ steps. To succeed, the agent must choose action 1 in $N - 1$ consecutive steps. The function `chain` builds this MDP as a `d2l.TabularMDP(P, r, gamma)`, the arrays of Section 14.1: `P[s, a]` is the distribution of the next state and `r[s, a]` the expected reward. The function `success_random` computes the exact probability that uniformly random actions reach the end within `horizon` steps. It averages the kernel over the two actions, $P_\pi(s' \mid s) = \frac{1}{2} \sum_a P(s' \mid s, a)$, and propagates the state distribution from state 0 through P_π one step at a time. The last state is absorbing, so its probability after `horizon` steps is the probability of having arrived:

```
def chain(N, gamma):
    """N states in a row: action 1 moves one state on, action 0 returns to
    the start; entering the last state pays 1 and ends the episode."""
    P, r = np.zeros((N, 2, N)), np.zeros((N, 2))
    P[:, 0, 0] = 1
    P[np.arange(N - 1), 1, np.arange(1, N)] = 1
    P[N - 1] = 0
    P[N - 1, :, N - 1] = 1              # the last state is absorbing
    r[N - 2, 1] = 1
    return d2l.TabularMDP(P, r, gamma)

def success_random(mdp, horizon):
    """Exact probability that uniformly random actions, started in state 0,
    reach the last state within `horizon` steps."""
    P_pi, d = mdp.P.mean(axis=1), np.eye(mdp.num_states)[0]
    for _ in range(horizon):
        d = d @ P_pi
    return d[-1]
```

`d2l.q_learning(seed, Q, visits, env, num_episodes, gamma, alpha, epsilon)` of Section 14.3 updates the table `Q` in place from transitions it collects with ϵ-greedy actions, counts visits in `visits`, and yields the return of each episode. It touches the environment only through `env.reset` and `env.step`, and the class `ModelEnv` gives a `TabularMDP` these two methods: `reset` puts the agent in state 0, and `step` draws the next state from `P[s, a]`, reporting termination in a terminal state and truncation after `max_steps` steps.

```
class ModelEnv:
    """Gymnasium's reset/step on a TabularMDP: start in state 0, draw s'
    from P[s, a], end in a terminal state or after max_steps steps."""
    def __init__(self, mdp, terminal, max_steps):
        self.mdp, self.terminal, self.max_steps = mdp, terminal, max_steps
        self.rng = np.random.default_rng(0)

    def reset(self, seed=None):
        if seed is not None:
            self.rng = np.random.default_rng(seed)
        self.s, self.t = 0, 0
        return self.s, {}

    def step(self, a):
        s, self.t = self.s, self.t + 1
        self.s = int(self.rng.choice(self.mdp.num_states, p=self.mdp.P[s, a]))
        return (self.s, self.mdp.r[s, a], self.s in self.terminal,
                self.t >= self.max_steps, {})
```

The learner is the Q-learning of Section 14.3 with one change: every entry of the table starts at 1, an upper bound on every return, and the agent acts greedily, $\epsilon = 0$. This is *optimistic initialization*, the idea behind UCB applied to a table: an untried pair looks as good as anything can be. For comparison, the cell prints the expected number of episodes until random actions first succeed, the inverse of the probability from `success_random`. Q-learning from a zero table acts exactly like this until its first success, whatever its ϵ, because every row is a tie broken at random (exercise 1):

```
def first_success(env, gamma, seed, max_episodes=100):
    """Episodes until greedy Q-learning from an optimistic table first
    reaches the goal (inf if it never does within max_episodes)."""
    # Optimistic: every entry starts at 1, an upper bound on every return
    Q = np.ones((env.mdp.num_states, env.mdp.num_actions))
    returns = d2l.q_learning(seed, Q, np.zeros_like(Q), env, max_episodes,
                             gamma, epsilon=lambda episode: 0.0)
    return next((k + 1 for k, ret in enumerate(returns) if ret > 0), np.inf)

for N in (4, 8, 12, 16, 20):
    mdp = chain(N, gamma=0.95)
    env = ModelEnv(mdp, terminal={N - 1}, max_steps=2 * N)
    first = [first_success(env, mdp.gamma, seed) for seed in range(20)]
    print(f'N = {N:2d}: episodes to the first success: random '
          f'{1 / success_random(mdp, 2 * N):5.0f} (mean), optimistic '
          f'{np.median(first):2.0f} (median, max {max(first):.0f})')
```

```
N =  4: episodes to the first success: random     2 (mean), optimistic  2 (median, max 2)
N =  8: episodes to the first success: random    23 (mean), optimistic  3 (median, max 4)
N = 12: episodes to the first success: random   273 (mean), optimistic  4 (median, max 5)
N = 16: episodes to the first success: random  3449 (mean), optimistic  6 (median, max 7)
N = 20: episodes to the first success: random 45590 (mean), optimistic  8 (median, max 10)
```

Random actions need about twice as many episodes for every state added: 23 at $N = 8$, about $3{,}400$ at $N = 16$, about $46{,}000$ at $N = 20$. The optimistic learner needs a handful, about eight at $N = 20$, a number that grows roughly in proportion to N. Optimism works because a tried pair is marked as tried. After one try of a pair that pays nothing, its entry moves toward a target $r + \gamma \max_{a'} Q(s', a')$ of at most $\gamma < 1$, so it falls below every untried entry, which still reads 1. The greedy rule therefore prefers a pair it has not tried, and the learner works through the chain systematically instead of by chance. The number of episodes grows polynomially in N instead of exponentially.

14.4.3.2 Count-Based Bonuses

Optimistic initialization explores a pair until its first try lowers its value. In a stochastic MDP one try says little, and a pair needs repeated visits before its value is known. *Count-based bonuses* carry the UCB rule over to MDPs. They add to the reward of each pair a bonus that shrinks like $1/\sqrt{n(s, a)}$ with its visit count, and let the value backups carry the bonus back to the states that lead there, so the agent plans its way toward pairs it knows little about. Model-based interval estimation does this with a learned model (Strehl and Littman 2008), and Q-learning with such bonuses has a regret that grows like the square root of the number of steps, up to factors polynomial in the size of the MDP (Jin et al. 2018). With a network in place of a table, states are rarely visited twice, and practical agents replace the count by a learned measure of novelty, such as the error of a network trained to predict a fixed random function of the state (Burda et al. 2019).

14.4.3.3 Coverage on the Lake

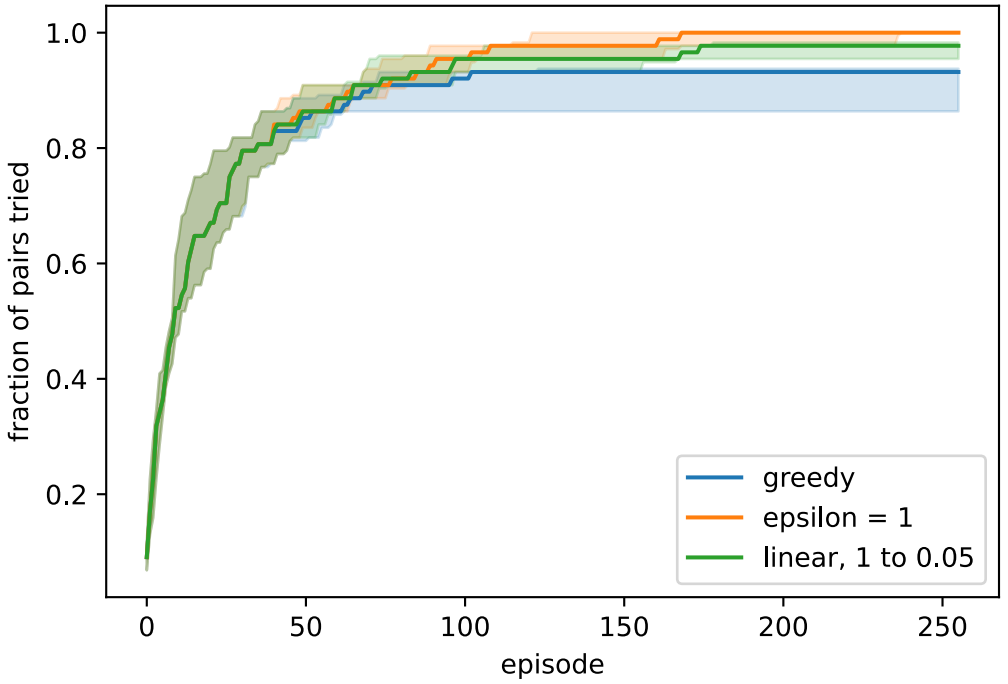
Does the lake need any of this? In Section 14.3, Q-learning on the calm lake, where every move goes where it is commanded, used the step size $\alpha = 0.9$ for 256 episodes, and its greedy table solved the lake. We repeat that run under three schedules: greedy, $\epsilon = 0$; always random, $\epsilon = 1$; and the default of `q_learning`, `d2l.linear_schedule(1.0, 0.05, 128)`,

which lowers ϵ linearly from 1 to 0.05 over the first half of training. In each of twenty runs the cell records, after every episode, the fraction of the 44 pairs tried at least once (11 cells to act from, four actions each). It also records the fraction of training episodes that reach the goal, and whether the greedy policy of the final table does, which `d2l.evaluate` of Section 14.2 checks in one episode on this deterministic lake:

```
def coverage_run(env, live, epsilon, seed, num_episodes=256, gamma=0.95):
    """One run of Q-learning with step size 0.9: the fraction of live pairs
    tried after each episode, the fraction of episodes won, and whether the
    final greedy policy reaches the goal."""
    Q = np.zeros((env.observation_space.n, env.action_space.n))
    visits, tried, returns = np.zeros_like(Q), [], []
    for ret in d2l.q_learning(seed, Q, visits, env, num_episodes, gamma,
                              alpha=lambda n: 0.9, epsilon=epsilon):
        tried.append((visits[live] > 0).mean())  # visits is updated in place
        returns.append(ret)
    solved = d2l.evaluate(env, lambda s, rng: int(Q[s].argmax()), 1)
    return tried, np.mean(returns), solved

calm = gym.make('FrozenLake-v1', is_slippery=False)
live = np.array([c not in b'HG' for c in calm.unwrapped.desc.flatten()])
schedules = {'greedy': lambda k: 0.0,
             'epsilon = 1': lambda k: 1.0,
             'linear, 1 to 0.05': d2l.linear_schedule(1.0, 0.05, 128)}
num_seeds, coverage = 20, {}
for name, epsilon in schedules.items():
    tried, won, solved = zip(*[coverage_run(calm, live, epsilon, seed)
                               for seed in range(num_seeds)])
    coverage[name] = np.array(tried)                # (seed, episode)
    print(f'{name:>17}: pairs tried {np.median(coverage[name][:, -1]):.0%}, '
          f'episodes won {np.median(won):.0%} (medians); greedy policy '
          f'solves {np.sum(solved):.0f}/{num_seeds}')
d2l.plot_curves(coverage, xlabel='episode', ylabel='fraction of pairs tried',
                band='iqr')
```

```
          greedy: pairs tried 93%, episodes won 60% (medians); greedy policy solves 19/20
     epsilon = 1: pairs tried 100%, episodes won 2% (medians); greedy policy solves 19/20
linear, 1 to 0.05: pairs tried 98%, episodes won 58% (medians); greedy policy solves 19/20
```



On the calm lake, random tie-breaking alone makes the greedy rule explore: it tries about nine pairs in ten, and its final greedy table solves the lake in nearly every run, as do the tables of the other two schedules. With $\epsilon = 1$ the agent tries every pair but reaches the goal in almost none of its training episodes. Learning a good table and collecting reward while learning are different goals, and ϵ trades one for the other. What limits all three schedules is the first success. Until then every entry is zero and every schedule acts at random, so for a given seed the first success arrives in the same episode under every schedule. The goal is six moves from the start, and random walks find it within a few dozen episodes. The lake is too small to show the difficulty of the chain. On slippery ice, the lake of this section's opening, a single visit to a pair says little about it, and trying every pair once is not enough (exercise 1).

14.4.4 Exploration on the Verifier Task

The verifier task of Section 14.1 has four arithmetic prompts x, nine candidate responses y shared by all of them, and a verifier with $r(x, y) = 1$ if y is the prompt's answer and 0 otherwise. The reference policy π_{ref}, a softmax over one row of preferences per prompt, spreads its probability evenly over the eight ordinary candidates. It gives almost none to the ninth, a hedge that lists every answer and that the exact-match verifier never accepts. Two prompts, $2+2$ and 3×4, have their answer among the candidates; for 8×8 and 9×9 no candidate matches. A sampled response is correct with probability 0.062 on average over the prompts.

Each round a prompt arrives, one response is scored, and the episode ends. The prompt is a context that the agent observes but cannot influence, and the response is the arm. A bandit whose best arm depends on an observed context is a *contextual bandit*: the agent faces one bandit per prompt and, with a model, can share what it learns across prompts.

A language model explores such a task differently from the rules above. A common way to train it, derived in Section 14.7, samples a group of K responses to a prompt from the current policy, scores them, and raises the probability of the responses that scored above the group's average while lowering those below it. A group in which every response scores the same has nothing above or below its average and changes nothing. If the policy answers a prompt correctly with probability $p = \pi(\text{correct} \mid x)$, a group of K independent responses is identical with probability

$$(1-p)^K + p^K, \tag{14.42}$$

the chance that all K are wrong plus the chance that all are right. The cell samples $1{,}000$ groups per prompt and compares the fraction of identical groups with this formula. It does so at the reference policy and at a policy that has nearly solved the two answerable prompts, built by hand by raising the preference of each correct answer by 5; it is not a trained policy. The array `R[x, y]` holds the verifier's score of every pair:

```
task = d2l.VerifierTask()
R, rng = task.R, np.random.default_rng(0)     # R[x, y] = r(x, y)
for name, pi in [('reference', task.pi_ref),
                 ('solved', task.policy(task.theta_ref + 5 * R))]:
    p = (pi * R).sum(axis=1)            # chance that one response is right
    for K in (2, 8, 32):
        same = []                       # per prompt: all K rewards equal
        for x in range(len(task.prompts)):
            ys = rng.choice(len(task.resp), (1000, K), p=pi[x])
            same.append((R[x, ys] == R[x, ys[:, :1]]).all(axis=1).mean())
        print(f'{name:>9}, K = {K:2d}: identical {np.round(same, 2)}, '
              f'mean {np.mean(same):.2f}, '
              f'exact {np.mean((1 - p) ** K + p ** K):.2f}')
```

```
reference, K =  2: identical [0.78 0.78 1.   1.  ], mean 0.89, exact 0.89
reference, K =  8: identical [0.35 0.33 1.   1.  ], mean 0.67, exact 0.67
reference, K = 32: identical [0.01 0.02 1.   1.  ], mean 0.51, exact 0.51
   solved, K =  2: identical [0.89 0.93 1.   1.  ], mean 0.96, exact 0.96
   solved, K =  8: identical [0.67 0.7  1.   1.  ], mean 0.84, exact 0.85
   solved, K = 32: identical [0.21 0.21 1.   1.  ], mean 0.60, exact 0.61
```

The sampled fractions agree with the formula. Half of all groups are identical at every K: on the two prompts without a correct candidate every response fails, and no amount of sampling finds an answer that the policy cannot produce. On the two answerable prompts the reference is right with probability $p \approx 0.125$, and larger groups are more often mixed: about 78% of the groups of two are identical, a third of the groups of eight, and almost none of the groups of 32. Once a prompt is nearly solved, groups become identical again, about two thirds of them at $K = 8$, now because almost every response is right, and learning on that prompt slows.

The learner explores only through its own sampling. The probability that a group contains at least one correct response, $1 - (1-p)^K$, is the *pass@K* score used to evaluate models for code and reasoning (M. Chen et al. 2021). On a prompt that the policy usually fails, it is nearly the probability that a group can teach anything. Sampling at a higher temperature spreads probability over more responses and mixes more groups on a prompt the policy

has nearly solved, at the cost of more wrong answers. It cannot help where no candidate is right.

Every rule in this section learns which actions are good by trying them, and the trying costs pulls of worse arms, episodes of random walking, or groups that teach nothing. On the chain an untried pair is hard to reach, and on the verifier task a correct answer that the policy never produces is never found. An expert who shows good actions would spare the agent all of this, and Section 14.5 learns from such demonstrations.

14.4.5 Summary

An agent that learns from its own actions must try actions that look worse to find out whether they are, and regret measures what this costs against an agent that knew the best action from the start. On a ten-armed bandit, the greedy rule settles on the first arm that pays out, and a fixed ϵ loses the same amount per pull forever. An ϵ that decays like $1/t$, UCB and Thompson sampling make regret grow only like the logarithm of the number of pulls, which Lai and Robbins showed is the best possible rate. UCB's coefficient decides its regret on a given problem: the coefficient of the theory is safe on every bandit and slower on this one. In an MDP, random actions find a required sequence of actions with a probability that falls geometrically in its length, while optimistic values make the agent seek out untried pairs. When a verifier scores groups of sampled responses, groups that score alike carry no signal, and the policy's own sampling is its only exploration.

14.4.6 Exercises

1. code **The exploration schedule.** ▯▯ On a zero-initialized table, `epsilon_greedy` breaks every tie uniformly at random, and every reward of FrozenLake is nonnegative.
    1. Show that every entry of the table stays exactly zero until the first successful episode, and that until then every ϵ produces the same behavior. What does this imply for the episode of the first success as a criterion for choosing ϵ?
    2. After the first success, which entries are positive? Show that an entry, once positive, stays positive, and that the greedy rule, $\epsilon = 0$, never again tries the zero entries of a row that contains a positive entry. The coverage experiment of this section shows this rule trying about nine pairs in ten on the calm lake and learning to succeed there. Predict whether it also learns a good policy on slippery ice.
    3. The optional argument `epsilon` of `q_learning` is a function from the episode index to the exploration rate, such as those returned by `linear_schedule`. On the slippery lake, with the 4000 episodes of Section 14.3, run five seeds each with ϵ held at 0, 0.1, 0.5 and 1, and with the schedule `linear_schedule(1.0, 0.05, num_episodes // 2)` that `q_learning` uses in Section 14.3. For each setting, report the mean training return, the success rate of the final greedy policy under `d2l.evaluate` over 1000 episodes, the fraction of pairs tried, and $\max_s |\max_a \hat{Q}(s,a) - V^*(s)|$. Was your prediction right?
    4. Rank the settings by training return and by greedy success rate. Where do the two rankings disagree, and what explains the disagreement?
2. **Logarithmic regret of UCB.** ▯▯ With $\kappa = \sqrt{2}$, the rule Equation 14.41 is UCB1 of Auer et al. (2002), whose Theorem 1 bounds the expected regret after t pulls of arms with

rewards in $[0, 1]$ by

$$\sum_{a:\,\Delta_a>0} \frac{8 \ln t}{\Delta_a} + \Big(1 + \frac{\pi^2}{3}\Big) \sum_a \Delta_a, \qquad \Delta_a = \mu^* - \mu_a. \tag{14.43}$$

Derive this bound. The tool is Hoeffding's inequality: the mean $\hat{\mu}_n$ of n independent rewards in $[0, 1]$ with mean μ satisfies

$$P(\hat{\mu}_n \geq \mu + h) \leq e^{-2nh^2}, \tag{14.44}$$

and likewise for the lower tail.

1. Fix a suboptimal arm a and let $n_0 = \lceil 8 \ln t / \Delta_a^2 \rceil$. Show that if arm a is pulled at a step $u \leq t$ while $n(a) \geq n_0$, then either the index of the best arm lies at or below μ^*, or the empirical mean of arm a exceeds μ_a by more than its bonus.
2. For fixed pull counts of both arms, bound the probability of each of the two events at step u by u^{-4}. Take a union bound over the counts and sum over u to show $\mathbb{E}[n(a)] \leq n_0 + \pi^2/3$ after t pulls.
3. Combine part (b) with the regret decomposition Equation 14.40 to obtain the theorem. Evaluate the bound for `arms` at $t = 2000$ and compare it with the median regret near 200 that the coefficient sweep of this section prints for $\kappa = \sqrt{2}$. Which step of the proof loses the most?
4. Repeat the argument for a general coefficient κ. For which κ does the sum over steps in part (b) still converge, and what does the answer say about the section's choice $\kappa = 0.5$? (The union bound over counts is loose; a peeling argument of (Bubeck and Cesa-Bianchi 2012) reaches every $\kappa > 1$.)

3. **Thompson's posterior.** ▯ Thompson sampling draws each arm's mean from Beta$(1 + w, 1 + n - w)$ after w wins in n pulls.
   1. Show that this is the posterior of a Bernoulli mean under the uniform prior on $[0, 1]$.
   2. Arm A has 3 wins in 4 pulls and arm B is untried. Show that Thompson sampling pulls B with probability $1/3$. More generally, show that against an untried arm it pulls the untried one with probability one minus the posterior mean of the other.
   3. Show that, given the data, Thompson sampling pulls each arm with the posterior probability that this arm has the largest mean.
   4. code Replace the uniform prior by Beta$(1, 9)$, which expects arms to be bad, and by Beta$(9, 1)$, which expects them to be good. How does the median regret after 2000 pulls change? Which of the two priors acts like optimistic initialization, and why?

4. code **Long horizons.** ▯▯ `bandit_run` plays one run at a time.
   1. Write a version that plays many runs at once: keep `wins` and `count` as arrays of shape (runs, arms) and choose the arms of all runs in one step, for UCB and for Thompson sampling. Check that its medians at 2000 pulls agree with those of this section up to the spread over runs.
   2. Run 200 runs of UCB with $\kappa \in \{0.15, 0.3, 0.5, \sqrt{2}\}$ and of Thompson sampling to 10^5 pulls, and plot the median regret on logarithmic axes. Does $\kappa = \sqrt{2}$ catch up with $\kappa = 0.5$?

3. For $\kappa = 0.15$, plot the mean regret as well as the median. Why do they grow at different rates, and which of the two describes the risk to a user who runs the rule once?

5. [extended] **Deep exploration.** ▯▯ Take the chain of this section: states $0, \dots, N-1$, start in state 0, action 0 returns to state 0, action 1 advances one state, reaching state $N-1$ gives reward 1 and terminates, and episodes are truncated after $2N$ steps.
   1. Show that the uniformly random policy succeeds in one episode with probability at most $2N \cdot 2^{-(N-1)}$, so that the expected number of episodes up to the first success is at least $2^{N-2}/N$, and compare the bound with the exact values that the chain cell prints. By part (a) of exercise 1, ϵ-greedy Q-learning from a zero table acts uniformly at random until its first success, so the bound holds for it whatever the value of ϵ.
   2. Trace the first episodes of the optimistic greedy learner on the chain with $N = 4$: which entries does each episode lower? Use the pattern to argue that the number of episodes it needs grows at most polynomially in N.
   3. Move both learners to `gym.make('FrozenLake-v1', map_name='8x8', is_slippery=True)`, which truncates at 100 steps. Compare the optimistic greedy learner with the zero-initialized learner under the schedule `linear_schedule(1.0, 0.05, num_episodes // 2)` at 4000 and 20000 episodes, five seeds each. Report the success rate of the greedy policy under `d2l.evaluate`, the environment steps used, and, as a reference, the success rate of the optimal policy computed by value iteration. Does optimism keep its advantage on a stochastic map, and what does it cost?

   The project takes a few minutes of CPU time.

6. **Groups that teach nothing.** ▯ On a prompt that the policy answers correctly with probability p, a group of K independent responses is identical with probability $(1-p)^K + p^K$.
   1. For a fixed $K \geq 2$, which p makes a mixed group most likely, and how likely is it then?
   2. For small p, show that the smallest K for which a group is mixed with probability at least $1/2$ is about $\ln 2/p$. Evaluate it for the reference policy of the verifier task, $p \approx 0.125$, and for $p = 0.01$.
   3. The prompts of a training set have success probabilities p spread uniformly over $[0, 1]$. What fraction of the groups of size $K = 8$ is identical, and which prompts produce them?
   4. On the two prompts of `VerifierTask` without a correct candidate, every group is identical for every K and every sampling temperature. Name two changes, to the task or to the learner, that would let it learn on these prompts, and say what each needs that the verifier alone does not provide.

14.5 Learning from Demonstrations

Every model in the Basics part learned from labeled examples: an image and its class, a sentence and its next word. The previous section showed how much an agent loses when nobody supplies labels and it must find good actions by trying bad ones. Often someone can show us instead. Suppose we record what a skilled driver sees and does, the camera image at each moment and the steering angle chosen in response. Could we train a driving policy

from these pairs the way we trained a classifier on Fashion-MNIST, with the image as the input and the steering angle as the label?

Dean Pomerleau built such a network at Carnegie Mellon in the late 1980s. His network ALVINN, with a single hidden layer, mapped road images to a steering direction and drove a van along a road (Pomerleau 1989); a later version learned to drive in a few minutes by watching a person do it (Pomerleau 1991). The method is now called *behavior cloning*. It needs neither a model of the environment nor a reward, only an expert whose choices can be recorded, and it is the most common way to learn from an expert today: fine-tuning a language model on curated responses is behavior cloning, with the text so far as the state and the next token as the action.

Pomerleau also saw its weakness. The driver kept the van in the middle of the lane, so the recordings contained no view of the roadside and no example of steering back. Once the network drifted, it saw images it had never been shown, and he had to synthesize shifted views of the road, labeled with the steering that would bring the van back. A classifier is tested on inputs like those it was trained on, but a policy chooses the inputs it sees next. This section measures what that costs on the lake, shows how the cost can grow with the length of the task, and changes the data so that the expert labels the states the learner reaches on its own.

```
import gymnasium as gym
```

14.5.1 Behavior Cloning

A *demonstration* is a trajectory recorded from an expert. Behavior cloning keeps only the states the expert visited and the actions it took there; it uses no rewards and no transition model. On the lake, the expert is the optimal policy π^* that value iteration computes in Section 14.2, so in this section π^* names both the optimal policy and the expert. We use calm ice, where every command moves the agent in the intended direction. The expert then walks the same six-move path from the start to the goal in every episode, and its states form a narrow tube through the lake, like the middle of the lane in Pomerleau's recordings. Because the model of the lake is known, we can compute every success rate exactly and compare the learner with the expert without sampling noise.

14.5.1.1 The Reduction to Classification

The cell below builds the calm lake as a `TabularMDP`, the arrays P and r with the discount γ of Section 14.1, solves it with `d2l.value_iteration`, and reads the expert off `mdp.backup`, the Bellman operator applied to V^*, by taking the best action in each state. `demonstrations` rolls the expert in the environment and records only what it saw and what it did:

```
gamma = 0.95
env = gym.make('FrozenLake-v1', is_slippery=False)
mdp = d2l.TabularMDP.from_gym(env, gamma)
V_star = d2l.value_iteration(mdp, num_iters=1000)[-1]
pi_star = mdp.backup(V_star).argmax(axis=1)

def demonstrations(env, expert, num_episodes):
    """Roll the expert; record only what it saw and what it did."""
```

```
    states, actions = [], []
    for _ in range(num_episodes):
        s, done = env.reset()[0], False
        while not done:
            states.append(s)
            actions.append(int(expert[s]))
            s, _, terminated, truncated, _ = env.step(actions[-1])
            done = terminated or truncated
    return np.array(states), np.array(actions)

demo_s, demo_a = demonstrations(env, pi_star, 3)
print(f'{demo_s.size} state-action pairs from 3 expert episodes, '
      f'covering states {np.unique(demo_s)}')
```

```
18 state-action pairs from 3 expert episodes, covering states [ 0  4  8  9 13 14]
```

Three episodes give 18 labeled pairs, the same six pairs three times, because on calm ice every expert episode is the same: *down*, *down*, *right*, *down*, *right*, *right*. At states 0 and 9, *down* and *right* are equally short, and the tie goes to *down*. The expert never visits the frozen cells 1, 2, 3, 6 and 10.

Behavior cloning treats these pairs as Section 3.1 treated images and labels: the state is the input, the expert's action is the class, and we fit a conditional distribution $\pi_\theta(a \mid s)$ with parameters θ by maximizing the log-likelihood $\sum_i \log \pi_\theta(a_i \mid s_i)$, that is, by minimizing the cross-entropy. All we need is a policy with parameters and a way to take gradient steps on it.

14.5.1.2 The Policy Container

`Policy` holds a network `policy` that maps a state to one score per action, so that $\pi_\theta(\cdot \mid s)$ is the softmax of the scores at s, together with an Adam optimizer `opt_pi` for its parameters. `log_prob` returns $\log \pi_\theta(a \mid s)$ for a batch of states and the actions taken there, the quantity that behavior cloning maximizes:

```
class seeded:
    """In a `with seeded(seed):` block, PyTorch's CPU generator starts from
    seed; on exit it resumes its previous state, so the seed affects only
    the draws inside the block. seed=None draws from the global stream."""
    def __init__(self, seed):
        self.seed = seed

    def __enter__(self):
        if self.seed is not None:
            self.state = torch.get_rng_state()
            torch.default_generator.manual_seed(self.seed)

    def __exit__(self, *exc):
        if self.seed is not None:
            torch.set_rng_state(self.state)

class Policy(nn.Module):
    """A policy network and its optimizer; pi(.|s) is the softmax of the
```

```
    network's outputs at s."""
    def __init__(self, policy, lr=1e-2):
        super().__init__()
        self.policy = policy
        self.opt_pi = torch.optim.Adam(policy.parameters(), lr=lr)

    def log_prob(self, obs, act):
        """log pi(a|s) for a batch of states and the actions taken there."""
        return torch.log_softmax(self.policy(obs), dim=-1) \
                    .gather(-1, act[:, None]).squeeze(-1)

    @classmethod
    def tabular(cls, num_states, num_actions, lr=0.1, seed=None):
        """One preference theta_{s,a} per state-action pair: an embedding."""
        with seeded(seed):
            policy = nn.Embedding(num_states, num_actions)
        nn.init.zeros_(policy.weight)
        return cls(policy, lr)
```

The `tabular` constructor is a softmax-regression model: an embedding table holds one preference $\theta_{s,a}$ per state–action pair, an integer state indexes its row, and a softmax produces $\pi_\theta(\cdot \mid s)$. Zero initialization makes the initial policy uniform.

The argument `seed` fixes the random initialization of the networks that this and later constructors build. `seeded` starts PyTorch's CPU generator from `seed` inside a `with` block and restores the generator's state afterwards, so the seed affects no random draw outside the constructor.

The environment exchanges NumPy integers and arrays, so three methods convert at the boundary. `act` samples an action from $\pi_\theta(\cdot \mid s)$ with a NumPy generator that the caller passes in, so all randomness that reaches the environment comes from one visible source. It follows the acting protocol of `d2l.evaluate` from Section 14.2, which averages the returns of sampled episodes. `act_greedy` returns the most probable action, and `log_prob_np` returns log-probabilities as a NumPy array:

```
@d2l.add_to_class(Policy)
def act(self, obs, rng):
    """Sample an action; numpy in, int out, the acting protocol of evaluate."""
    with torch.inference_mode():
        probs = torch.softmax(self.policy(torch.as_tensor(obs)), -1).numpy()
    return int(rng.choice(len(probs), p=probs))

@d2l.add_to_class(Policy)
def act_greedy(self, obs, rng=None):
    """The most probable action; the same protocol as act."""
    with torch.inference_mode():
        return int(self.policy(torch.as_tensor(obs)).argmax())

@d2l.add_to_class(Policy)
def log_prob_np(self, obs, act):
    """log pi(a|s) for numpy batches of states and actions, as numpy."""
    with torch.inference_mode():
```

```
        return self.log_prob(torch.as_tensor(obs),
                             torch.as_tensor(act)).numpy()
```

The three methods run under `torch.inference_mode`, because nothing they compute is differentiated.

For a policy over a small set of states, `probs` reads the whole table $\pi_\theta(a \mid s)$ off the learner as an array, for the exact computations of this chapter:

```
@d2l.add_to_class(Policy)
def probs(self, num_states, num_actions):
    """pi(a|s) for every pair, as a (num_states, num_actions) numpy array."""
    s, a = np.divmod(np.arange(num_states * num_actions), num_actions)
    return np.exp(self.log_prob_np(s, a)).reshape(num_states, num_actions)
```

14.5.1.3 Fitting the Clone

Fitting the clone is gradient ascent on $\frac{1}{n}\sum_i w_i \log \pi_\theta(a_i \mid s_i)$ with every weight w_i equal to one. `policy_step` takes one optimizer step on this objective for a batch of states, actions and weights:

```
def policy_step(ac, obs, act, weight):
    """One ascent step on (1/n) sum_t w_t log pi(a_t|s_t) over a batch of
    n states and actions; the weights w arrive as numpy, that is, as data."""
    w = torch.as_tensor(weight)
    loss = -(w * ac.log_prob(torch.as_tensor(obs),
                             torch.as_tensor(act))).mean()
    ac.opt_pi.zero_grad()
    loss.backward()
    ac.opt_pi.step()
    return loss.item()
```

The weights arrive as a NumPy array, that is, as data, and no gradient flows into them. For behavior cloning they are all one.

How long should the fit run? Trained to convergence, a table with one row per state would copy the six demonstrated actions exactly, and on calm ice the clone would then walk the expert's path every time. Real classifiers are not exact: inputs that look alike, limited capacity and noisy labels leave some error on the training distribution. To give the clone such an error, `clone` calls `policy_step` with unit weights until π_θ disagrees with the demonstrated actions with probability at most $\varepsilon = 0.2$, averaged over the pairs. The stopping rule fixes ε, so every comparison in this section holds the fit error constant:

```
def clone(mdp, states, actions, eps=0.2):
    """Behavior cloning: cross-entropy fit of pi(a|s) to the expert's
    choices (policy_step with every weight equal to one), stopped once pi
    disagrees with the demonstrated actions with probability at most eps."""
    learner = Policy.tabular(mdp.num_states, mdp.num_actions)
    weight = np.ones(len(states), np.float32)
    while 1 - np.exp(learner.log_prob_np(states, actions)).mean() > eps:
```

```
            policy_step(learner, states, actions, weight)
        return learner

    bc = clone(mdp, demo_s, demo_a)
    pi_bc = bc.probs(mdp.num_states, mdp.num_actions)
    for s in (9, 10):
        print(f'clone pi(.|s={s}): {np.round(pi_bc[s], 3)}')
```

```
clone pi(.|s=9): [0.061 0.818 0.061 0.061]
clone pi(.|s=10): [0.25 0.25 0.25 0.25]
```

Fourteen Adam steps bring the disagreement to 0.182. Every demonstrated state appears equally often and receives the same gradient, so each of the six rows ends with probability 0.818 on the expert's action and 0.061 on each of the three others, as the row of state 9 shows. State 10 was never demonstrated. Its row receives no gradient and stays uniform, so there the clone disagrees with the expert three times in four. Repeating the data does not change the fit: the mean loss over three copies of six pairs is the mean over one copy, so a clone of one expert episode and a clone of a hundred are the same table.

14.5.2 Distribution Shift

14.5.2.1 Training and Deployment

A classifier's guarantee refers to the distribution its training data came from. Behavior cloning trains on the states the expert visits and is deployed on the states its own actions produce. We deploy the clone in two ways: greedily, taking the most probable action, and by sampling from π_θ with `act`, as a language model samples its next token.

Because the lake is known, the state distribution of a policy can be computed instead of sampled. Let $d_t^\pi(s)$ be the probability that policy π is in state s after t steps. It starts with all mass on the start state, and one step moves the mass through the policy and the kernel:

$$d_{t+1}^\pi(s') = \sum_s d_t^\pi(s) \sum_a \pi(a \mid s)\, P(s' \mid s, a). \tag{14.45}$$

The mass in s' at the next step is the mass that each state sends there under the actions the policy chooses. `visitation` carries out this recursion for the lake's 100-step time limit; the goal and the holes end an episode and keep their mass, so the mass at the goal after 100 steps is the probability of success. A Monte Carlo run of `d2l.evaluate` checks the exact number for the sampling clone:

```
def visitation(mdp, pi, absorbing=None, T=100):
    """d_0, ..., d_T: the state distributions of the policy pi[s, a] from the
    start state, propagated exactly through P for FrozenLake's 100-step
    limit. Mass that enters an `absorbing` state stays there."""
    P_pi = np.einsum('sa,sat->st', pi, mdp.P)
    if absorbing is not None:
        P_pi[absorbing] = np.eye(mdp.num_states)[absorbing]
    d = [np.eye(mdp.num_states)[0]]
    for _ in range(T):
        d.append(d[-1] @ P_pi)
```

```
        return np.array(d)

    desc = env.unwrapped.desc.ravel()
    goal, holes = desc == b'G', desc == b'H'
    expert = np.eye(mdp.num_actions)[pi_star]                # one-hot rows
    greedy = np.eye(mdp.num_actions)[pi_bc.argmax(axis=1)]
    for name, pi in [('expert', expert), ('greedy clone', greedy),
                     ('sampling clone', pi_bc)]:
        d_T = visitation(mdp, pi)[-1]
        print(f'{name:>14}: reaches the goal {d_T[goal].sum():.3f}, '
              f'falls into a hole {d_T[holes].sum():.3f}')
    rate = d2l.evaluate(env, bc.act, num_episodes=1000,
                        rng=np.random.default_rng(0))
    print(f'sampling clone, 1000 episodes of bc.act: reaches the goal {rate:.3f}')
```

```
       expert: reaches the goal 1.000, falls into a hole 0.000
  greedy clone: reaches the goal 1.000, falls into a hole 0.000
sampling clone: reaches the goal 0.660, falls into a hole 0.340
sampling clone, 1000 episodes of bc.act: reaches the goal 0.656
```

The expert reaches the goal every time, and so does the greedy clone: the most probable action of each demonstrated row is the expert's, so the greedy clone walks the expert's path. The sampling clone reaches the goal in 66% of its episodes and falls into a hole in the rest; the 1,000 sampled episodes agree within their noise. Nothing on calm ice is random except the clone's own choices, so every step it takes off the path is one of its own wrong actions.

Is the classifier to blame? The next cell measures how often the clone disagrees with the expert on three sets of states: its training pairs, the states of 100 fresh expert episodes, and the states the clone visits itself, weighted by how often it visits them. It then clones 1, 10 and 100 expert episodes:

```
    disagree = 1 - pi_bc[np.arange(mdp.num_states), pi_star]   # per state
    fresh_s, _ = demonstrations(env, pi_star, 100)
    occupancy = (visitation(mdp, pi_bc)[:-1] * ~(goal | holes)).sum(axis=0)
    print('disagreement with the expert')
    print(f'  on the training pairs:        {disagree[demo_s].mean():.3f}')
    print(f'  on 100 fresh expert episodes: {disagree[fresh_s].mean():.3f}')
    print(f"  on the clone's own states:    "
          f'{occupancy @ disagree / occupancy.sum():.3f}')
    for n in [1, 10, 100]:
        pi_n = clone(mdp, *demonstrations(env, pi_star, n)).probs(
            mdp.num_states, mdp.num_actions)
        print(f'clone of {n:>3} expert episodes: reaches the goal '
              f'{visitation(mdp, pi_n)[-1][goal].sum():.3f}')
```

```
disagreement with the expert
  on the training pairs:        0.182
  on 100 fresh expert episodes: 0.182
  on the clone's own states:    0.216
clone of   1 expert episodes: reaches the goal 0.660
clone of  10 expert episodes: reaches the goal 0.660
clone of 100 expert episodes: reaches the goal 0.660
```

On fresh expert episodes the clone errs as often as on its training pairs, 18% of the time, so there is no generalization gap: the demonstrations cover the expert's distribution. Here more expert data cannot help: it adds no new state, and every clone of the repeated six pairs is the same table, succeeding 66% of the time. On its own states the clone errs more often, 22% of the time, because some of them are cells where it has no data and disagrees with probability $3/4$. The states the policy is tested on have moved, and its own actions moved them. This is *distribution shift*. On a six-step task the shift is small, and Section 14.5.3.2 measures how much of the loss it causes.

14.5.2.2 Missing Coverage

Missing coverage is a different failure. On slippery ice the expert itself slides, so the set of states it visits is random, and a few episodes can miss a state where the right action matters. The next cell builds the slippery lake and its optimal policy, draws 200 datasets of n expert episodes for each n, and evaluates the greedy clone of each dataset exactly. That clone takes the expert's action at every demonstrated state; at every other state its row is uniform and the tie goes to action 0. The cell writes this table down directly instead of fitting 600 clones:

```
ice = gym.make('FrozenLake-v1', is_slippery=True)
mdp_ice = d2l.TabularMDP.from_gym(ice, gamma)
V_ice = d2l.value_iteration(mdp_ice, num_iters=1000)[-1]
pi_ice = mdp_ice.backup(V_ice).argmax(axis=1)
one_hot = np.eye(mdp_ice.num_actions)
expert_ice = visitation(mdp_ice, one_hot[pi_ice])[-1][goal].sum()
ice.reset(seed=5)
print(f'share of 200 datasets whose greedy clone is below half the expert '
      f'({expert_ice:.3f}):')
for n in [1, 3, 10]:
    rates = []
    for _ in range(200):
        seen = np.isin(np.arange(mdp_ice.num_states),
                       demonstrations(ice, pi_ice, n)[0])
        # the greedy clone: the expert's action where it has data; elsewhere
        # its row is untouched and uniform, and the tie goes to action 0
        tab = np.where(seen, pi_ice, 0)
        rates.append(visitation(mdp_ice, one_hot[tab])[-1][goal].sum())
    failed = np.mean(np.array(rates) < expert_ice / 2)
    print(f'  expert episodes per dataset {n:>2}: {failed:.1%}')
```

```
share of 200 datasets whose greedy clone is below half the expert (0.730):
  expert episodes per dataset  1: 32.0%
  expert episodes per dataset  3: 1.5%
  expert episodes per dataset 10: 0.0%
```

With one episode, about one dataset in three gives a clone below half the expert's success rate; with three episodes, one or two in a hundred; with ten, none of the 200. Missing coverage disappears as expert data accumulate. Distribution shift does not, because the states the clone reaches through its own mistakes are states the expert never visits.

14.5.3 Compounding Error

14.5.3.1 Quadratic and Linear Error Growth

The expert's path on the lake is six moves long. How does the cost of a per-step error grow when the task is longer? Supervised learning bounds the probability of an error at one decision. In a sequential task an early error changes all later states and can cost all later rewards. Demonstrations sample states from the expert's distributions $d_t^{\pi^*}$, so behavior cloning controls the clone's error rate there and nowhere else.

Proposition (compounding error). *Consider an episodic task of T steps with rewards in $[0, 1]$ at every step, and a policy $\hat{\pi}$ that at every step t disagrees with the expert with probability at most ε when the state is drawn from the expert's distribution $d_t^{\pi^*}$. (i) The expected return of $\hat{\pi}$ falls short of the expert's by at most $\varepsilon T(T+1)/2 \le \varepsilon T^2$, and there are tasks in which the shortfall is of this order whenever $\varepsilon T \le 1$. (ii) If instead $\hat{\pi}$ disagrees with probability at most ε when the state is drawn from its own distribution $d_t^{\hat{\pi}}$, and one wrong action followed by the expert's behavior costs at most u, the shortfall is at most $u\varepsilon T$* (Ross and Bagnell 2010).

Proof. (i) Run $\hat{\pi}$ and the expert from the same start with the same environment randomness, so that their trajectories coincide until $\hat{\pi}$ first takes an action the expert would not take. The expert's state at step t is drawn from $d_t^{\pi^*}$, and the first disagreement can fall at step t only if $\hat{\pi}$ disagrees at that state, which has probability at most ε. From then on $\hat{\pi}$ can lose at most the rewards of steps $t, \dots, T$, which number $T - t + 1$. Summing over t, the shortfall is at most $\sum_{t=1}^{T} \varepsilon(T - t + 1) = \varepsilon T(T+1)/2$. (ii) Let π_k follow $\hat{\pi}$ for the first k steps and the expert afterwards, so that π_0 is the expert and π_T is $\hat{\pi}$. Two consecutive hybrids π_{k-1} and π_k differ only in the action at step k, taken at a state drawn from $d_k^{\hat{\pi}}$. That action disagrees with the expert with probability at most ε, and a disagreement followed by the expert's behavior costs at most u. The expert's return minus that of $\hat{\pi}$ is the sum of the T differences between consecutive hybrids, hence at most $u\varepsilon T$. ■

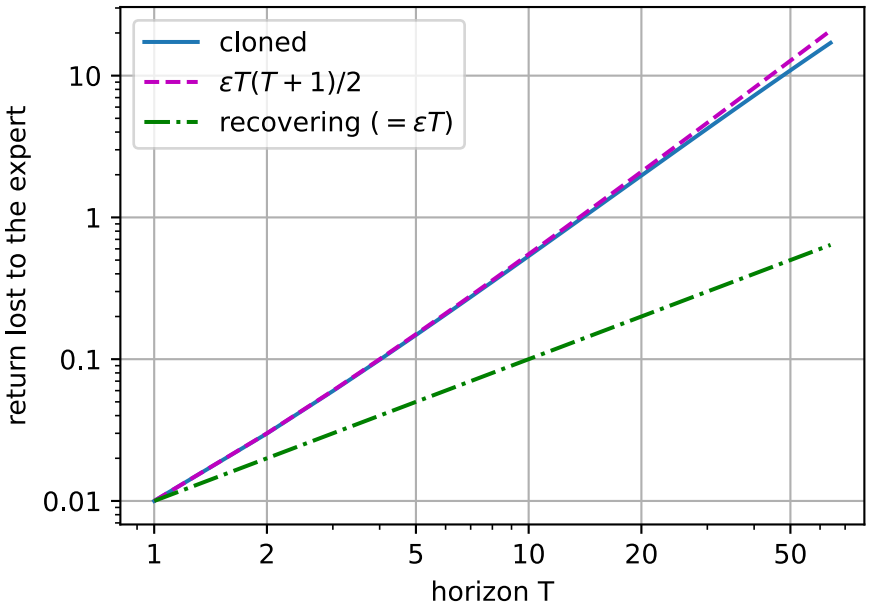
The second part is the telescoping argument of Kakade and Langford (2002) . If ε is the average of the per-step error rates over the T steps rather than their maximum, the sum in part (i) is still at most εT^2. Part (i) charges a first mistake with everything that follows, because afterwards the policy is in states where nothing controls its error. Part (ii) moves the measurement of ε to the policy's own states, and it is useful only when u is small, that is, when the task allows recovery from a single wrong action. If one mistake is fatal, u is as large as the remaining horizon and part (ii) gives order εT^2 again.

The horizon of the lake is fixed, so we watch the growth with T on a chain of T steps instead. The expert moves right at every step and earns one per step. Two imitators disagree with it independently with probability ε at every step. For the *cloned* imitator a mistake is permanent: it leaves the chain for states it was never shown and earns nothing afterwards, so it earns the reward of step t with probability $(1-\varepsilon)^t$. The *recovering* imitator returns to the chain after one step, so a mistake costs exactly one reward, part (ii) with $u = 1$, and it earns the reward of step t with probability $1 - \varepsilon$. The cell computes the lost return exactly for $\varepsilon = 0.01$ and T up to 64:

```
eps, T_max = 0.01, 64
T = np.arange(1, T_max + 1)
# probability of earning the reward of step t: the cloned imitator only if
# it has agreed at every step up to t, the recovering one if it agrees at t
```

```
earn_bc = np.cumprod(np.full(T_max, 1 - eps))
earn_rec = np.full(T_max, 1 - eps)
gap_bc, gap_rec = T - earn_bc.cumsum(), T - earn_rec.cumsum()
d2l.plot([T] * 3, [gap_bc, eps * T * (T + 1) / 2, gap_rec],
         xlabel='horizon T', ylabel='return lost to the expert',
         legend=['cloned', '$\\varepsilon T(T+1)/2$',
                 'recovering ($=\\varepsilon T$)'],
         xscale='log', yscale='log', figsize=(5, 3.5))
print(f'lost return at T={T_max}: cloned {gap_bc[-1]:.2f} '
      f'(eps T(T+1)/2 = {eps * T_max * (T_max + 1) / 2:.2f}), '
      f'recovering {gap_rec[-1]:.2f} (eps T = {eps * T_max:.2f})')
```

```
lost return at T=64: cloned 17.03 (eps T(T+1)/2 = 20.80), recovering 0.64 (eps T = 0.64)
```



On logarithmic axes the cloned imitator stays close to the quadratic reference $\varepsilon T(T+1)/2$, within 3% at $T = 10$ and 10% at $T = 30$, and falls gradually below it as εT grows, because a first mistake can happen only once: the first disagreement falls at step t with probability $\varepsilon(1-\varepsilon)^{t-1}$, less than the ε the bound charges. It never rises above the reference, as part (i) requires. The recovering imitator loses exactly εT. At $T = 64$ the cloned imitator loses about 17 rewards and the recovering one 0.64, about 27 times less. Exercise 1 derives both curves in closed form.

14.5.3.2 Decomposition of the Clone's Loss

Back on the lake, how much of the clone's 34-point loss comes from distribution shift? The clone can fail in two ways. It can fall into a hole straight from the expert's path, through a wrong action at one of the six states it was trained on; that is the error ε on the expert's distribution, which only a better fit removes. Or it can leave the path for a cell where it has no data and fail from there; that is the shift. The next cell makes the cells off the path absorbing, so that `visitation` measures how often the clone leaves the path, and compares

how often the episodes that stay on it and those that leave it reach the goal. Its last line evaluates a learner with the clone's 18% error at every cell, on the path and off it:

```
path = np.isin(np.arange(mdp.num_states), demo_s)    # the expert's cells
off_path = ~path & ~(goal | holes)
d_T = visitation(mdp, pi_bc, absorbing=off_path)[-1]
left = d_T[off_path].sum()
stay_goal = d_T[goal].sum()
clone_goal = visitation(mdp, pi_bc)[-1][goal].sum()
print(f'the clone leaves the expert path in {left:.3f} of its episodes, '
      f'entering cells {np.flatnonzero(d_T * off_path)}')
print(f'it reaches the goal in {stay_goal / (1 - left):.3f} of the episodes '
      f'that stay on the path, {(clone_goal - stay_goal) / left:.3f} of '
      f'those that leave it')
# the clone's demonstrated rows, with their error, at every cell
p_agree = pi_bc[demo_s[0], demo_a[0]]
labeled = np.where(expert == 1, p_agree, (1 - p_agree) / 3)
print(f'a learner with the same error at every cell reaches the goal '
      f'{visitation(mdp, labeled)[-1][goal].sum():.3f}')
```

```
the clone leaves the expert path in 0.172 of its episodes, entering cells [ 1 10]
it reaches the goal in 0.715 of the episodes that stay on the path, 0.394 of those that leave it
a learner with the same error at every cell reaches the goal 0.735
```

The clone leaves the path in 17% of its episodes, always through a wrong action at state 0 into cell 1, or at state 9 or 14 into cell 10 (Figure 14.11). Episodes that stay on the path reach the goal 72% of the time, and episodes that leave it 39% of the time. The loss splits accordingly. Of the 34 points, 24 are falls from the path itself (0.828×0.285) and 10 are failures after leaving it (0.172×0.606). A learner with the clone's error at every cell would reach the goal 73.5% of the time, so labels off the path can recover about 7.5 of the 34 points, less than the 10, because a labeled cell still errs 18% of the time. On this lake distribution shift accounts for about a fifth of the clone's loss. The rest is the 18% error itself, on the path and off it, which no change of data removes. On a longer path the clone leaves it far more often; exercise 6 finds what that does to the two parts of the loss on the 8×8 lake.

14.5.4 Interactive Data Collection

14.5.4.1 Dataset Aggregation

More demonstrations cannot fix the shift, because the expert never visits the cells where the clone goes wrong. The learner has to visit them, and the expert has to label them. *DAgger*, short for dataset aggregation, does this (Ross et al. 2011): it runs the current learner, asks the expert which action it would have taken at each state the learner visited, adds these pairs to the dataset, and refits. The model, the loss, the optimizer and the stopping rule stay as they are; only the data change.

`relabel` rolls the learner with `act`, sampling as it does when deployed, and labels its states with the expert's actions:

```
def relabel(env, expert, learner, num_episodes, rng):
    """Roll the learner, sampling its actions; label its states with the
```

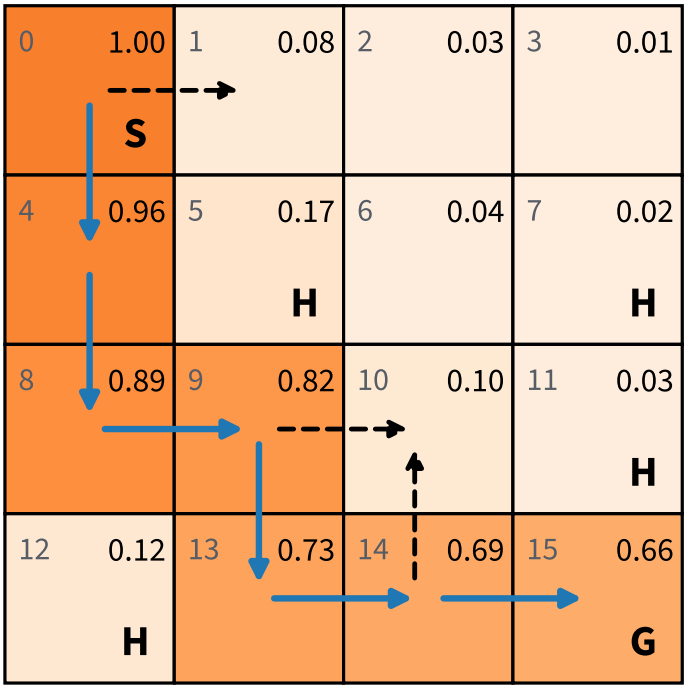


Figure 14.11: Where this section's clone goes on the calm lake. Blue arrows mark the expert's six-move path. Each cell shows the probability that the deployed clone, which samples its actions and disagrees with the expert with probability $\varepsilon = 0.18$ on the expert's states, enters the cell within the 100-step limit; darker cells are entered more often. The dashed arrows are the clone's exits from the path into cells where it has no data and acts uniformly: from 0 into 1, and from 9 or 14 into 10. The clone reaches the goal with probability 0.66; the holes beside the path, 5 and 12, take most of the rest.

```
    expert's actions."""
    visited = []
    for _ in range(num_episodes):
        s, done = env.reset()[0], False
        while not done:
            visited.append(s)
            s, _, terminated, truncated, _ = env.step(learner.act(s, rng))
            done = terminated or truncated
    visited = np.array(visited)
    return visited, expert[visited]
```

`dagger` starts from the three demonstrations, runs four rounds of ten learner episodes each, refits with `clone` after every round, and evaluates each learner exactly. `d2l.run_seeds` from the online appendix "Utility Functions and Classes" runs it for ten seeds and stacks the results:

```
def dagger(seed, env, mdp, expert, demo_s, demo_a, num_rounds=4,
           num_episodes=10):
    """Yield (labels, distinct states, success) for behavior cloning and
    each DAgger round after it."""
    rng = np.random.default_rng(seed)
    goal = env.unwrapped.desc.ravel() == b'G'
    S, A = demo_s, demo_a                       # the aggregated dataset
    for k in range(num_rounds + 1):
        learner = clone(mdp, S, A)
        pi = learner.probs(mdp.num_states, mdp.num_actions)
        yield S.size, np.unique(S).size, visitation(mdp, pi)[-1][goal].sum()
        if k < num_rounds:            # the learner visits, the expert labels
            new_s, new_a = relabel(env, expert, learner, num_episodes, rng)
            S, A = np.concatenate([S, new_s]), np.concatenate([A, new_a])

runs = d2l.run_seeds(dagger, num_seeds=10, env=env, mdp=mdp, expert=pi_star,
                     demo_s=demo_s, demo_a=demo_a)   # (seed, round, quantity)
for k, (n, m, rate) in enumerate(np.median(runs, axis=0)):
    print(f'round {k}: {n:>3.0f} labels at {m:>2.0f} states, success '
          f'{rate:.3f} (seeds: {runs[:, k, 2].min():.3f} to '
          f'{runs[:, k, 2].max():.3f}; medians over 10 seeds)')
```

```
round 0:  18 labels at  6 states, success 0.660 (seeds: 0.660 to 0.660; medians over 10 seeds)
round 1:  84 labels at  8 states, success 0.710 (seeds: 0.649 to 0.735; medians over 10 seeds)
round 2: 152 labels at  9 states, success 0.717 (seeds: 0.652 to 0.735; medians over 10 seeds)
round 3: 218 labels at 11 states, success 0.735 (seeds: 0.710 to 0.735; medians over 10 seeds)
round 4: 283 labels at 11 states, success 0.735 (seeds: 0.717 to 0.735; medians over 10 seeds)
```

Round 0 is behavior cloning at 0.66. Each round adds about 65 labels, most of them at the six states of the path, and now and then a cell off it. Within four rounds, a few hundred labels, the median over ten seeds reaches 0.70 to 0.735, depending on which ten, at or near the 0.735 of a learner with the clone's error at every cell; on every seed the learner of the fourth round is above behavior cloning. Behavior cloning with any number of expert labels stays at 0.66. DAgger recovers the part of the loss that came from states off the path. The remaining gap to the expert is the 18% error of the fit itself, which DAgger does not touch. A learner lands on the 0.735 exactly, not approximately: Adam divides each parameter's step

by the size of that parameter's own gradients, and the gradient of a labeled row is its label count times a factor common to all labeled rows, so every labeled row ends at the same 0.818 however often it was labeled. The learner is therefore fixed by which cells carry labels, and once every cell it reaches has one, it is the 0.735 learner.

The rounds need not improve monotonically. On some seeds a round lowers success slightly, to 0.649 at worst, and exercise 8 finds the cause. The guarantee of Ross et al. (2011) concerns the best learner of the sequence, or one drawn from it at random, not the last one. It assumes many rounds of data collection, a no-regret learner trained on the accumulated dataset and, in the original analysis, a mixture of expert and learner actions during collection. It also needs the expert on call to label whatever states the learner reaches. Keeping the data of all earlier rounds prevents the fitted policy from depending only on its most recent trajectories.

14.5.4.2 Supervised Fine-Tuning as Behavior Cloning

The token MDP of Section 14.1 makes supervised fine-tuning of a language model an instance of behavior cloning. The state is the context, the prompt with the tokens written so far; the action is the next token; the transition appends it with probability one; and the curated responses are the expert's demonstrations. The cross-entropy of fine-tuning is the negative log-likelihood of behavior cloning, summed over the tokens of each response. The horizon T is the length of the response, and u is the cost of one wrong token if the model continues well afterwards: little for an unusual but harmless word, a lot for a wrong digit early in a calculation that every later step builds on. At deployment the model samples its own tokens and conditions on them, so its contexts drift from those of the curated responses as a response grows.

DAgger for a language model needs an expert who labels the model's own prefixes: for each context the student generated, which token should come next. People cannot do this for millions of prefixes, but a larger model can. On-policy distillation trains a student on its own samples, with the next-token distributions of a teacher model as the labels (Agarwal et al. 2024); it is DAgger with the teacher as the expert on call. Behavior cloning is also the standard first baseline in offline reinforcement learning, where only a fixed dataset is available, and Section 15.7 compares offline methods against it.

Behavior cloning turns an expert's recorded choices into a policy with nothing more than a classifier, and DAgger removes the part of its loss that comes from the learner's own states by changing the data alone. Both need an expert: one who demonstrates, and for DAgger one on call to label whatever the learner does. Without an expert, the cloning update still applies if the weight $w_i = 1$ is replaced by a number that says how well the action turned out. Section 14.6 derives that weight from the reward.

14.5.5 Summary

Behavior cloning fits a policy $\pi_\theta(a \mid s)$ to an expert's state–action pairs by cross-entropy, as softmax regression, and needs neither a transition model nor a reward; each fitting step is a gradient step on log-probabilities with unit weights. The fit controls the policy's error on the expert's state distribution, but the deployed policy visits the states its own actions produce. A per-step error ε on the expert's states can cost up to $\varepsilon T(T+1)/2$ of return over T steps, while the same error on the policy's own states costs at most $u\varepsilon T$ when one mistake costs at most u. On the six-move path of the calm lake, more expert data cannot change the clone, and the shift to states the expert never visits accounts for about a fifth of its loss. DAgger

labels the learner's own states with the expert's actions and recovers that part by changing only the data.

14.5.6 Exercises

1. **The chain in closed form.** In the chain experiment of Section 14.5.3.1, an imitator disagrees with the expert independently with probability ε at each step $t = 1, \dots, T$. The cloned imitator earns the reward of step t only if it has agreed at every step up to t. The recovering imitator earns it whenever it agrees at step t.
   1. Derive in closed form the expected return $G(T)$ that the cloned imitator loses to the expert, and show that the recovering imitator loses εT. Compare both with the values printed for $\varepsilon = 0.01$ and $T = 64$.
   2. Expand $G(T)$ to second order in ε. Show that the first-order term is the bound $\varepsilon T(T+1)/2$ of the proposition, and find the second-order term. What does its sign say about where the cloned curve lies relative to the reference? For $\varepsilon = 0.01$, up to which horizon is the first-order term within 10% of $G(T)$?
   3. How does $G(T)$ grow once $\varepsilon T \gg 1$, and which condition on εT does the quadratic regime therefore require? Compute the ratio of the two losses at $\varepsilon = 0.01$ and $T = 64$, the right end of the plot, and compare it with the ratio $(T+1)/2$ of the two references.
2. code **Where the clone's mass goes.** Use `visitation` to propagate the state distributions of the expert and of the sampling clone `pi_bc` over the 100 steps of the time limit.
   1. Plot the total variation distance $\frac{1}{2}\sum_s |d_t^{\pi^*}(s) - d_t^{\hat{\pi}}(s)|$ against t, together with the mass that the clone places on the goal, on each hole, and on the cells off the expert's path. Which holes take the clone's lost successes? Around $t = 6$ the total variation exceeds one half, although the mass off the path stays at about 5 percent. What else makes the two distributions differ? Consider what a wrong action does at states 0, 4 and 8.
   2. Show that $|\mathbb{E}_{s\sim d}[f(s)] - \mathbb{E}_{s\sim d'}[f(s)]|$ is at most the total variation distance between d and d' for every function f with values in $[0,1]$, with equality for the indicator of the set where d exceeds d'. Conclude that the two success probabilities differ by at most the total variation distance at $t = 100$. Why does the bound hold with equality there, and why is the total variation at $t = 6$ a poor measure of distribution shift?
3. **Fine-tuning as behavior cloning.** Section 14.5.4.2 identifies supervised fine-tuning of a language model with behavior cloning: the context is the state, the next token is the action, and the curated responses are the demonstrations.
   1. The proposition of Section 14.5.3.1 assumes the same error rate ε at every step. A fine-tuned model may err at a small constant rate along curated responses, yet at a rate that grows with the position along its own responses. Explain why, and say which part of the proposition then describes the model.
   2. Apart from their number, why are people a poor source of DAgger's labels for a language model? On-policy distillation labels each of the student's prefixes with the teacher's whole next-token distribution rather than with one token. Write down the resulting loss, and show that it reduces to the behavior-cloning loss when the teacher puts all its probability on one token.
4. **Two valid paths.** A robot faces an obstacle. Half of the demonstrations pass it on the left and half on the right. Each maneuver takes several steps, during which the robot's observations of the two maneuvers barely differ.
   1. A deterministic policy is fitted to the demonstrated steering angles $-c$ and $+c$ by

squared error. Which angle does it output in front of the obstacle?
2. The steering is discretized into *left*, *straight*, and *right*, and a categorical policy is fitted by cross-entropy, as `clone` does. What does the fit converge to in front of the obstacle, and what does `act_greedy` do there?
3. Diffusion policies (C. Chi et al. 2023) sample a sequence of several actions from one observation and execute it before sampling again. Compare this with sampling one action per step with `act` during the maneuver. What can go wrong with the per-step sampler, and what does executing a sequence without replanning cost?

5. code **Demonstrations needed for coverage.** ⊠ The coverage cell of Section 14.5.2 builds the slippery lake `mdp_ice` and its optimal policy `pi_ice`. The probability that a deterministic policy `pi` reaches the goal is the mass at the goal after `visitation(mdp_ice, one_hot[pi])`.
   1. For each nonterminal state s, compute the success probability of the policy that follows `pi_ice` except at s, where it takes action 0. Call s critical if this deviation lowers the success probability to less than half that of `pi_ice`. Which states are critical?
   2. Let q_s be the probability that one expert episode visits s. Compute q_s exactly with the `absorbing` argument of `visitation`: make s absorbing and read off the mass at s after 100 steps. Which nonterminal state has $q_s = 0$, and what does that imply for any amount of expert data? How probable is a dataset of three episodes that never visits state 13?
   3. Treating the critical states as independent, predict from their q_s the probability that $n \in \{1, 3, 10\}$ expert episodes leave some critical state undemonstrated, and compare the prediction with the rates that the cell prints. Is the prediction too high or too low, and what does that say about visits to different critical states?
6. code **DAgger on a longer path.** ⊠ On the 4×4 calm lake, distribution shift accounts for about a fifth of the clone's loss. Repeat the experiment on the 8×8 calm lake, `gym.make('FrozenLake-v1', map_name='8x8', is_slippery=False)`, whose optimal path takes 14 moves: rebuild `env`, `mdp`, `pi_star`, `demo_s` and `demo_a`, and rerun the cells of this section.
   1. Compute the sampling clone's success probability, the probability that it leaves the expert's path, and the success probability of a learner with the clone's error at every cell. Which share of the clone's loss is incurred after it leaves the path, and which share can labels off the path recover? Why does the first grow with the length of the path while the second does not?
   2. Run DAgger for ten seeds and plot the success rate against the number of expert queries, next to behavior cloning on at least as many expert labels.
   3. A tabular policy needs at most one label per state. Count the distinct states that DAgger has labeled after each round, and estimate how many queries it would need if it asked the expert only at states without a label.
7. code **A noisy expert.** Suppose that the expert reports a uniformly random action instead of its own with probability 0.1, in the demonstrations and in DAgger's labels alike.
   1. To which distribution does cross-entropy converge at a state labeled many times by this expert? What does `act_greedy` do there, and what does `act` do?
   2. Corrupt the labels of `demonstrations` and `relabel` in this way, rerun `clone` and DAgger on the calm lake, and compare their exact success rates with those of the section. Against which actions does the stopping rule of `clone` now measure the disagreement, and how does that change what $\varepsilon = 0.2$ means?

3. Which part of the clone's loss grows with the noise, the falls from the path or the failures after leaving it? Which part can DAgger still remove?

8. **DAgger's guarantee.** ⊠ In the analysis of Ross et al. (2011), the data-collecting policy π_i of iteration i takes the expert's action with probability β_i at each step and the action of the current learner $\hat{\pi}_i$ otherwise. The next learner $\hat{\pi}_{i+1}$ minimizes the loss on the aggregated data of iterations $1, \dots, i$, as refitting with `clone` does. Write $d_\pi = \frac{1}{T}\sum_{t=1}^{T} d_t^\pi$ for the average state distribution of π, let $c(s,\pi) \in [0, c_{\max}]$ be a classification loss of π at s, such as $\mathbf{1}(\pi(s) \neq \pi^*(s))$, and set $c_i(\pi) = \mathbb{E}_{s\sim d_{\pi_i}}[c(s,\pi)]$.
   1. Show that $\|d_{\pi_i} - d_{\hat{\pi}_i}\|_1 \leq 2\min(1, T\beta_i)$.
   2. Let $\varepsilon_N = \min_\pi \frac{1}{N}\sum_{i=1}^{N} c_i(\pi)$ be the loss of the best policy in hindsight, and suppose that the learners have average regret at most δ_N:

$$\frac{1}{N}\sum_{i=1}^{N} c_i(\hat{\pi}_i) - \varepsilon_N \leq \delta_N. \tag{14.46}$$

   Show that some learner $\hat{\pi}_j$ of the sequence satisfies

$$\mathbb{E}_{s\sim d_{\hat{\pi}_j}}[c(s,\hat{\pi}_j)] \leq \varepsilon_N + \delta_N + \frac{2c_{\max}}{N}\sum_{i=1}^{N} \min(1, T\beta_i). \tag{14.47}$$

   The proof of Theorem 4.1 of the paper derives this bound before it simplifies the last sum.
   3. The loop of Section 14.5.4.1 uses $\beta_1 = 1$ and $\beta_i = 0$ for $i \geq 2$. Specialize the bound to this schedule and to the 0-1 loss, and combine it with part (ii) of the proposition in Section 14.5.3.1. How must N grow with T for the term that does not involve ε_N or δ_N to stay bounded?
   4. On some seeds of the printed run, the learner of the first round is worse than behavior cloning. Show that labeling only state 1 lowers the exact success rate of the calm-lake clone, and explain the drop from the expert's action at state 1 and the learner's rows at states 1 and 2 before and after the label. What would identifying the best learner of a sequence require?

9. **Rewards behind demonstrations.** ⊠ The text learned from the expert's actions. *Inverse reinforcement learning* instead infers a reward under which the demonstrations are optimal and then plans with that reward.
   1. By the telescoping argument of Section 14.2.5, every potential-based shaping $\tilde{r}(s,a,s') = r(s,a) + \gamma\Phi(s') - \Phi(s)$ of r with $\Phi = 0$ at terminal states keeps the expert optimal. Show that the zero reward makes the expert optimal as well. What do the two facts imply about recovering r from demonstrations?
   2. Maximum-entropy inverse reinforcement learning (Ziebart et al. 2008) models the trajectories of a deterministic episodic task with a fixed start state as $P_\theta(\tau) \propto \exp\left(\sum_t r_\theta(s_t, a_t)\right)$ over the feasible trajectories, and fits θ by maximizing the likelihood of the demonstrations. Why does this criterion reject the zero reward for an expert that reaches the goal? Suppose every feasible trajectory ends at a terminal state. Why can the criterion then not distinguish r from its shaping with $\gamma = 1$, and which step of your argument fails for trajectories cut by a time limit?

14.6 Policy Gradient

In the previous section a policy learned from an expert. Behavior cloning took one ascent step on the log-probability of each demonstrated action, with every action weighted by one. Now suppose there is no expert, only the agent's own attempts and the reward each attempt earned. An obvious guess keeps the update and changes the weights: push up the log-probability of every action the agent took, weighted by the return of the episode it belongs to, so that actions from episodes that paid off become more likely. Is this guess principled? How does the expected return change when the parameters θ of the policy move?

Ronald Williams answered this question in 1992 (Williams 1992). The average of the guessed update is the gradient of the expected return, so the guess is stochastic gradient ascent, and he called the algorithm REINFORCE. The return of an episode depends on the environment's transition probabilities, which the agent does not know, yet the estimate needs only sampled episodes and the policy's own log-probabilities. Writing the policy down as a function $\pi_\theta(a \mid s)$ also lifts a limit of the value tables of Section 14.3. Acting greedily on a table needs a maximum over a finite list of actions, whereas a parameterized policy can output a torque, or a distribution over a vocabulary of tokens. The same identity, applied to the next-token distribution of a language model, trains reasoning models from a single reward given to the finished answer.

We derive the estimator from the smallest problem with a policy gradient and run it on the verifier task of Section 14.1, where every gradient can be computed exactly, and on the lake. It is right on average, but it needs fresh episodes after every update, and a small batch leaves much noise.

```
import gymnasium as gym
```

14.6.1 Parameterizing the Policy

Gradient ascent needs a policy that changes smoothly with its parameters. A deterministic policy is an argmax, which is piecewise constant: a small change of the parameters either changes nothing or switches the action. The probabilities of a stochastic policy can move by any small amount.

14.6.1.1 Softmax Preferences

`Policy.tabular` from Section 14.5 stores one parameter $\theta_{s,a}$, called a *preference*, for each state-action pair. A softmax over the actions turns the preferences at state s into probabilities:

$$\pi_\theta(a \mid s) = \frac{e^{\theta_{s,a}}}{\sum_{a'} e^{\theta_{s,a'}}}. \tag{14.48}$$

This is the softmax regression of Section 3.1 with a state index as its input, the same policy that behavior cloning fitted. The zero table gives the uniform policy. Unlike $Q(s, a)$, a preference is not an estimate of return; adding the same constant to every preference at a state leaves the policy unchanged.

14.6.1.2 The Score Function

The softmax is easy to differentiate. Since $\log \pi_\theta(a \mid s) = \theta_{s,a} - \log \sum_{a''} e^{\theta_{s,a''}}$, the derivative with respect to the preference $\theta_{s,a'}$ of any action a' at the same state is

$$\frac{\partial \log \pi_\theta(a \mid s)}{\partial \theta_{s,a'}} = \mathbf{1}(a' = a) - \pi_\theta(a' \mid s), \tag{14.49}$$

and the derivative with respect to the preferences at every other state is zero. The vector $\nabla_\theta \log \pi_\theta(a \mid s)$ is called the *score function*. In words, increasing the log-probability of a means raising its own preference and lowering every preference at s in proportion to its probability. Writing e_a for the indicator vector of action a, the row of the score at s is $e_a - \pi_\theta(\cdot \mid s)$. Its components sum to zero, so the score has no component along the one direction that leaves the policy unchanged, adding a constant to every preference at s. We check the formula against automatic differentiation of `Policy.log_prob` at a random table:

```
rng = np.random.default_rng(0)
check = d2l.Policy.tabular(16, 4)
with torch.no_grad():   # move to a generic table, off the uniform start
    check.policy.weight.copy_(torch.as_tensor(rng.standard_normal((16, 4))))
check.log_prob(torch.tensor([6]), torch.tensor([2])).backward()
probs = np.exp(check.log_prob_np(np.repeat(6, 4), np.arange(4)))
hand = np.zeros((16, 4), dtype=np.float32)
hand[6], hand[6, 2] = -probs, 1 - probs[2]
print(bool(torch.allclose(check.policy.weight.grad, torch.as_tensor(hand),
                          atol=1e-6)))
```

```
True
```

The gradient of the library's `log_prob` at state 6 and action 2 matches Equation 14.49, and it is zero in every other row of the table.

14.6.2 The Policy Gradient

What is the gradient of the expected return, and can it be estimated from the agent's own episodes? The answer is cleanest with one state, and it keeps its form for trajectories.

14.6.2.1 The Smallest Case

Take one state and two actions with preferences θ_1 and θ_2. Action 1 earns reward 1 and action 2 earns 0. Write $p = \pi_\theta(1)$. The expected reward is $J(\theta) = p$, and by Equation 14.49 its derivative with respect to θ_1 is $p(1-p)$. Suppose that we could not compute p but could only try actions. The derivative can still be written as an average over the policy's own actions. Multiply and divide each term by $\pi_\theta(a)$:

$$\frac{\partial J}{\partial \theta_1} = \sum_a r(a) \frac{\partial \pi_\theta(a)}{\partial \theta_1} = \sum_a \pi_\theta(a)\, r(a) \frac{\partial \log \pi_\theta(a)}{\partial \theta_1} = \mathbb{E}_{a \sim \pi_\theta}\left[r(a) \frac{\partial \log \pi_\theta(a)}{\partial \theta_1} \right].$$

So one sampled action, its reward multiplied by its score, is an estimate of the derivative. With probability p the sample is action 1, whose reward is 1 and whose score is $1 - p$; with

probability $1-p$ the reward is zero and so is the estimate. The mean is $p(1-p)$, the derivative itself. The second moment is $p(1-p)^2$, so the standard deviation is $(1-p)\sqrt{p(1-p)}$, which is

$$\sqrt{\frac{1-p}{p}} \tag{14.50}$$

times the mean. When the paying action is rare, the estimate is right on average and useless in almost every single draw: it is usually zero, and occasionally about $1/p$ times too large. Averaging n independent draws divides the ratio by $\sqrt{n}$.

The lake has the same structure at the start of learning. The uniform policy reaches the goal in about 1.4% of episodes (Section 14.2), on calm and slippery ice alike, because a uniformly random command is a uniformly random move. An episode that misses earns nothing. At $p = 0.014$ the ratio is 8.4, and a batch of 16 episodes contains no success at all with probability $(1-0.014)^{16} \approx 0.80$. Four batches in five then give an estimate of zero. The rest of this section carries out the same computation with trajectories in place of the two actions.

14.6.2.2 An Optimization Problem over Trajectories

On the lake an episode is a trajectory $\tau = (s_0, a_0, r_0, s_1, a_1, r_1, \dots, s_T)$ that ends at the first terminal state s_T, so its length T is random. The agent chooses each action and the environment chooses each next state, so the probability of a trajectory is a product of the policy's factors, the transition kernel's factors, and the start distribution μ_0 of Section 14.1:

$$P(\tau;\theta) = \mu_0(s_0) \prod_{t=0}^{T-1} \pi_\theta(a_t \mid s_t)\, P(s_{t+1} \mid s_t, a_t). \tag{14.51}$$

Only the policy's factors depend on θ. The objective is the expected return over these trajectories, with $R(\tau) = \sum_{t=0}^{T-1} \gamma^t r_t$:

$$J(\theta) = \mathbb{E}_{\tau \sim P(\cdot;\theta)}\big[R(\tau)\big] = \sum_{\tau} R(\tau)\, P(\tau;\theta). \tag{14.52}$$

The lake has a fixed start state, so $J(\theta) = V^{\pi_\theta}(s_0)$ in the notation of Section 14.2. On the verifier task μ_0 is the uniform distribution over the four prompts.

14.6.2.3 The Log-Derivative Trick

The step of the smallest case applies unchanged. The return $R(\tau)$ of a fixed trajectory does not depend on θ; only its probability does. Moving the gradient inside the sum and multiplying and dividing by $P(\tau;\theta)$ gives

$$\nabla_\theta J(\theta) = \sum_{\tau} R(\tau)\, \nabla_\theta P(\tau;\theta) = \sum_{\tau} P(\tau;\theta)\, R(\tau)\, \frac{\nabla_\theta P(\tau;\theta)}{P(\tau;\theta)} = \sum_{\tau} P(\tau;\theta)\, R(\tau)\, \nabla_\theta \log P(\tau;\theta),$$

that is,

$$\nabla_\theta J(\theta) = \mathbb{E}_{\tau \sim P(\cdot;\theta)}\Big[R(\tau)\, \nabla_\theta \log P(\tau;\theta)\Big]. \tag{14.53}$$

The identity $\nabla \log P = \nabla P / P$ is called the *log-derivative trick*. It turns the gradient back into an average over the trajectories that the current policy produces, which we can sample. The sum over trajectories is infinite, because an episode can last any number of steps. Moving the gradient inside is still allowed: under a softmax policy every state has a positive probability of reaching a terminal state within a few steps, so long episodes are exponentially rare, and each step adds a score of bounded length.

14.6.2.4 Cancellation of the Transition Probabilities

The gradient still contains $\log P(\tau;\theta)$, and $P(\tau;\theta)$ contains the transition kernel, which the agent does not know. The logarithm of the product Equation 14.51 is a sum, and only the policy's terms survive differentiation:

$$\begin{aligned}\nabla_\theta \log P(\tau;\theta) &= \nabla_\theta \Big[\log \mu_0(s_0) + \sum_{t=0}^{T-1} \log \pi_\theta(a_t \mid s_t) + \sum_{t=0}^{T-1} \log P(s_{t+1} \mid s_t, a_t)\Big] \\ &= \sum_{t=0}^{T-1} \nabla_\theta \log \pi_\theta(a_t \mid s_t).\end{aligned}$$

Figure 14.12 shows the chain of factors. The start distribution and the kernel do not depend on θ, so their logarithms have zero gradient. The kernel still decides which trajectories we sample, but the estimator never evaluates it. This is what makes the method *model-free*, and it names the assumption the derivation needs: the dynamics must not depend on the policy's parameters (exercise 1 drops it).

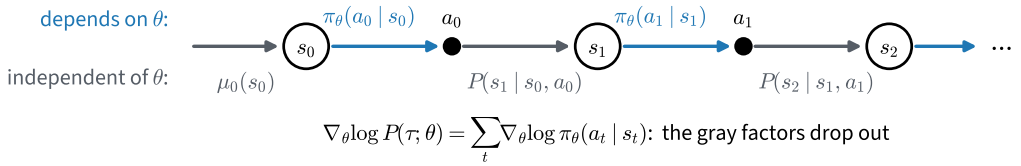


Figure 14.12: The probability of a trajectory as a chain of factors, $P(\tau;\theta) = \mu_0(s_0) \prod_t \pi_\theta(a_t \mid s_t)\, P(s_{t+1} \mid s_t, a_t)$. Only the policy factors (blue, above the chain) depend on θ. The start distribution and the transition kernel (gray, below) do not, so their logarithms have zero gradient, and the score of a trajectory is the sum of the policy's scores, which can be computed without knowing P.

14.6.2.5 The REINFORCE Estimator

Substituting the sum of scores into Equation 14.53 and replacing the expectation by an average over n trajectories $\tau_1, \dots, \tau_n$ sampled from the current policy gives

$$\hat{u} = \frac{1}{n} \sum_{i=1}^{n} R(\tau_i) \sum_{t=0}^{T_i - 1} \nabla_\theta \log \pi_\theta(a_t^i \mid s_t^i). \tag{14.54}$$

Each trajectory pushes up the log-probability of every action it took, by an amount proportional to its return. This is the guess of the opening, and each term has expectation $\nabla_\theta J(\theta)$, so $\hat{u}$ is an unbiased estimate of the gradient. REINFORCE (Williams 1992) takes the ascent step $\theta \leftarrow \theta + \eta\, \hat{u}$ with a step size η, then samples a new batch from the updated policy. On the lake every return is nonnegative, so every sampled action is pushed up. Actions that were not sampled lose probability through the $-\pi_\theta(\cdot \mid s)$ part of the score, and an episode with return zero changes nothing.

Figure 14.13 draws one step for a one-step problem with a continuous action, which a softmax over four actions cannot show as a curve. The policy is a Gaussian $\mathcal{N}(a; \mu, 1)$, whose score with respect to the mean is $a - \mu$. An episode is a single action a, so its return $R(a)$ is that action's reward, and the step reads $\mu \leftarrow \mu + \eta \frac{1}{n} \sum_i R(a_i)(a_i - \mu)$: the mean moves toward the sampled actions that paid most. Twelve samples move it to 0.32, where the exact gradient would give 0.52.

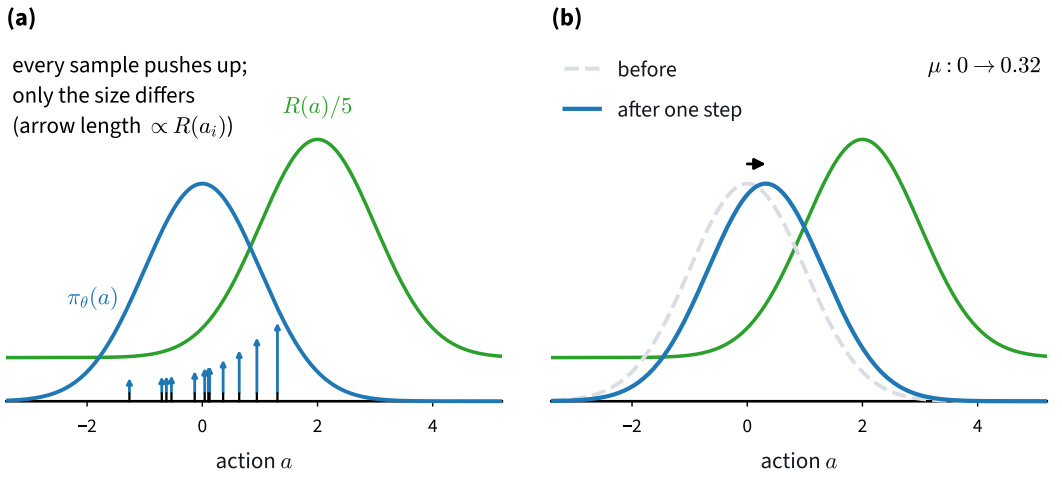


Figure 14.13: One REINFORCE step on a one-step problem. The policy is $\pi_\theta(a) = \mathcal{N}(a; \mu, 1)$ with $\mu = 0$, and the reward is $R(a) = 0.4 + 2e^{-(a-2)^2/2}$, so the expected reward is $J = 0.92$ and the exact gradient is $\mathrm{d}J/\mathrm{d}\mu = 0.52$. (a) Twelve sampled actions each push up their own log-probability, weighted by their rewards; all rewards are positive, so every arrow points up. (b) The step $\mu \leftarrow \mu + \eta \frac{1}{12} \sum_i R(a_i)(a_i - \mu)$ with $\eta = 1$ moves the mean from 0 to 0.32, toward the actions with higher reward; the exact gradient would have moved it to 0.52.

14.6.3 REINFORCE on the Running Examples

The verifier task has one step per episode and no transitions, so every quantity can be computed exactly. The lake has episodes of several steps and a transition kernel that the update never reads.

14.6.3.1 The Verifier Task

The verifier task of Section 14.1 has four arithmetic prompts x and nine candidate responses y shared by all of them. The policy is a table θ with one row of preferences per prompt, $\pi_\theta(y \mid x)$ is the softmax of row x, and the verifier pays $r(x, y) = 1$ for the correct answer and 0 otherwise. A trajectory is a single response to a prompt, with probability $\mu_0(x)\,\pi_\theta(y \mid x)$ and no kernel at all. Row x of θ affects only the expected score of prompt x, $J_x(\theta) = \sum_y \pi_\theta(y \mid x)\, r(x, y)$, and by Equation 14.49 with the prompt in place of the state, its exact gradient is a sum of nine terms:

$$\nabla_{\theta_x} J_x(\theta) = \sum_y \pi_\theta(y \mid x)\, r(x, y)\, (e_y - \pi_\theta(\cdot \mid x)).$$

REINFORCE replaces this sum by an average over K sampled responses. `verifier_step` samples K responses for every prompt, turns their rewards into weights with the function `weighting`, and moves the row of θ by `lr` $= \eta$ times the weighted average of the K scores. With `weighting = lambda r: r` the weights are the rewards and the step is Equation 14.54; Section 14.7 passes other weightings. The checker is the argument `reward`, so the same step can train against a different verifier.

```
def verifier_step(task, theta, K, reward, rng, weighting, lr=2.0):
    """One REINFORCE step on a VerifierTask: K responses per prompt, each
    score e_y - pi(.|x) (eq_softmax_score) weighted by weighting(rewards of
    the prompt's K responses), averaged over the K, one ascent step."""
    pi, g = task.policy(theta), np.zeros_like(theta)
    eye = np.eye(len(task.resp))
    for x in range(len(task.prompts)):          # one group of K per prompt
        ys = rng.choice(len(task.resp), K, p=pi[x])
        w = weighting(np.array([reward(x, y) for y in ys]))
        g[x] = (eye[ys] - pi[x]).T @ w / K
    return theta + lr * g
```

The exact-match verifier pays only the correct response y^*, so the estimate for prompt x is $\frac{c}{K}(e_{y^*} - \pi_\theta(\cdot \mid x))$, where c counts the correct responses among the K. The count c is binomial with success probability $p = \pi_\theta(y^* \mid x)$, so the estimate has mean $p\,(e_{y^*} - \pi_\theta(\cdot \mid x))$, the exact gradient, and relative standard deviation $\sqrt{(1-p)/(Kp)}$: the ratio of the smallest case divided by $\sqrt{K}$. At the reference table $p \approx 1/8$ on the two answerable prompts, so a group of $K = 16$ should be off by about 0.66 of the gradient's length.

The next cell checks this prediction. It draws 10,000 estimates at the reference table (with `lr=1.0` the step minus θ is the estimate) and takes one step with $\eta = 2$:

```
task = d2l.VerifierTask()
theta0, pi0, R = task.theta_ref, task.policy(task.theta_ref), task.R
exact = pi0 * R - pi0 * (pi0 * R).sum(axis=1, keepdims=True)  # nine terms
rng = np.random.default_rng(0)
U = np.stack([verifier_step(task, theta0, 16, task.verify, rng, lambda r: r,
                            lr=1.0) - theta0 for _ in range(10_000)])
err = lambda u: np.linalg.norm(u - exact) / np.linalg.norm(exact)
print(f'one K=16 estimate is off by '
```

```
      f'{np.sqrt(np.mean([err(u) ** 2 for u in U])):.2f} of the gradient (RMS)')
print('the mean of M of them: ' + ', '.join(
    f'M={M}: {err(U[:M].mean(axis=0)):.3f}' for M in (10, 100, 1000, 10_000)))
print(f'groups without a success, per prompt: '
      f'{np.round((U == 0).all(axis=-1).mean(axis=0), 2)}')
after = [task.success(task.policy(theta0 + 2.0 * u)) for u in U]
print(f'success {task.success(pi0):.3f}; one step with lr = 2 gives '
      f'{task.success(task.policy(theta0 + 2.0 * exact)):.3f} (exact gradient),'
      f' median {np.median(after):.3f} (sampled)')
```

```
one K=16 estimate is off by 0.66 of the gradient (RMS)
the mean of M of them: M=10: 0.126, M=100: 0.061, M=1000: 0.047, M=10000: 0.009
groups without a success, per prompt: [0.12 0.12 1.   1.  ]
success 0.062; one step with lr = 2 gives 0.077 (exact gradient), median 0.077 (sampled)
```

One estimate is off by 0.66 of the gradient's length, as predicted. The error of the mean of M estimates keeps falling as M grows, on average like $0.66/\sqrt{M}$, and shows no floor: the estimator is unbiased. Each estimate amounts to one binomial count per answerable prompt, so a single average can land well below that rate or up to about twice above it. On each answerable prompt about 12% of the groups contain no correct response, $(7/8)^{16} \approx 0.12$, and contribute nothing. On `8*8` and `9*9`, where no candidate passes the verifier, every group is empty: no amount of sampling produces a gradient for a prompt that no response solves. One exact step with $\eta = 2$ raises the success from 0.062 to 0.077, and a sampled step does about as well in the median. A sampled step never lowers the success on this task, because the only weighted score is that of the correct response.

14.6.3.2 The Lake

On the lake an episode has several steps and the kernel enters. We use the calm lake, where every command moves the agent as intended, rather than the slippery ice of the earlier sections. The reason is how much one episode says about one action. At the uniform start both maps give the same walk and the same success probability of about 1.4%. But on slippery ice a command moves the agent in its own direction or in either perpendicular direction with probability $1/3$ each, so the one thing it controls is that the agent does not move the opposite way. At the uniform start this makes the exact gradient on slippery ice one third as long as on calm ice, while the noise of a one-episode estimate is about the same, so a batch must be nine times larger for the same precision (exercise 4 derives the factor).

As the policy improves the gap widens: successful episodes on slippery ice last tens of steps instead of six, so each success is shared among many more actions and discounted by γ^{T-1}. REINFORCE still learns on slippery ice, slowly (exercise 7).

We remove Gymnasium's default limit of 100 steps for this map. The objective Equation 14.52 is defined over complete episodes, and the sum of rewards of a cut episode is not its return (Section 14.1). A softmax policy gives every action positive probability, so on this map every episode ends with probability one, and the cap of 10,000 steps is only a safeguard far above any episode these runs produce:

```
gamma = 0.95
env = gym.make('FrozenLake-v1', is_slippery=False, max_episode_steps=10_000)
```

`Batch` stores the transitions of several episodes in flat arrays, with the index one past the last step of each episode. `episode_returns` computes $R(\tau)$ for each episode, and `per_step` repeats one value per episode over that episode's steps, which turns a return into the weight of each of its actions. The container also records the next state and a flag `term` that marks *terminated*, never *truncated*: by the rule of Section 14.1, a cut episode has not reached a terminal state. Every lake episode here terminates, so the flag matters only once episodes are cut.

```
class Batch:
    """A flat batch of transitions, plus the episode boundaries."""
    def __init__(self, obs, act, rew, next_obs, term, ep_ends):
        self.obs, self.act, self.rew = obs, act, rew
        self.next_obs, self.term = next_obs, term
        self.ep_ends = ep_ends    # one past the last step of each episode

    def __len__(self):
        return len(self.rew)

    def episodes(self):
        """Yield one slice per episode."""
        start = 0
        for end in self.ep_ends:
            yield slice(start, end)
            start = end

    def episode_returns(self, gamma=1.0):
        """R(tau), the discounted return, one number per episode."""
        return np.array([(gamma ** np.arange(ep.stop - ep.start)
                          * self.rew[ep]).sum() for ep in self.episodes()])

    def per_step(self, values):
        """Repeat one value per episode over that episode's steps."""
        return np.repeat(values, np.diff(self.ep_ends, prepend=0))
```

`rollout` fills a batch. It plays complete episodes with any policy written as a function `policy(obs, rng) -> action`, the acting protocol of `evaluate` in Section 14.2, and appends each step's state, action, reward, next state and termination flag. All randomness of the agent comes from the NumPy generator `rng`, so the seed fixes every sampled episode.

```
def rollout(env, policy, num_episodes, rng):
    """Collect complete episodes from `policy(obs, rng) -> action` as a
    Batch. `term` records `terminated`, never `truncated`: a time-limit cut
    is not a terminal state. All sampling runs through the numpy `rng`."""
    cols, ep_ends = [[] for _ in range(5)], []
    for _ in range(num_episodes):
        obs, done = env.reset()[0], False
        while not done:
            act = policy(obs, rng)
            next_obs, reward, terminated, truncated, _ = env.step(act)
            done = terminated or truncated
            for col, val in zip(cols, (obs, act, reward, next_obs,
                                       float(terminated))):
                col.append(val)
```

```
            obs = next_obs
        ep_ends.append(len(cols[0]))
    obs, act, rew, next_obs, term = (np.asarray(c) for c in cols)
    return Batch(obs, act, rew.astype(np.float32), next_obs,
                 term.astype(np.float32), np.asarray(ep_ends))
```

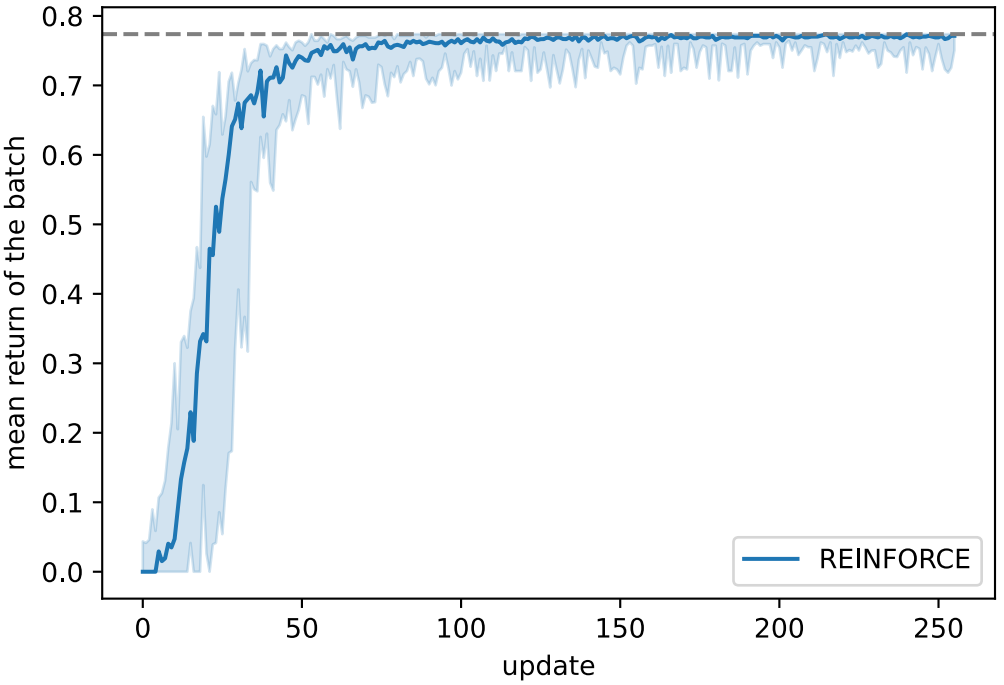
The training loop reuses `policy_step` from Section 14.5, which takes one optimizer step on $\frac{1}{N}\sum_t w_t \log \pi_\theta(a_t \mid s_t)$ over the N steps of a batch. Behavior cloning passed $w_t = 1$; REINFORCE passes the return of the episode that contains step t. Internally `policy_step` minimizes the negative of this sum, a *surrogate loss*, whose value says nothing about how well the policy performs; only its gradient at the current θ, the estimate, matters.

Two details differ from Equation 14.54. First, `policy_step` divides by the number of steps N rather than the number of episodes n, which divides the estimate by the mean episode length of the batch, a factor that varies from batch to batch. Second, the optimizer is Adam with the learning rate 0.1 of `Policy.tabular`, not the plain step $\theta \leftarrow \theta + \eta\hat{u}$. Plain ascent crawls wherever an action has small probability, because by Equation 14.49 the gradient of its preference carries that probability as a factor, while Adam moves each coordinate by about its learning rate (exercise 7 compares the two). The loop collects a fresh batch, weights each step by its episode's return, takes one step, and yields the batch's mean return and its number of steps:

```
def train_reinforce_tabular(ac, env, gamma, seed, num_updates=256,
                            batch_episodes=16):
    """REINFORCE: a fresh batch from the current policy, every step of a
    trajectory weighted by that trajectory's return, one ascent step.
    Yields the batch's mean return and its number of steps."""
    rng = np.random.default_rng(seed)      # one stream for all sampling
    env.reset(seed=seed)
    for _ in range(num_updates):
        batch = rollout(env, ac.act, batch_episodes, rng)
        R = batch.episode_returns(gamma)
        d2l.policy_step(ac, batch.obs, batch.act, batch.per_step(R))
        yield float(R.mean()), len(batch)
```

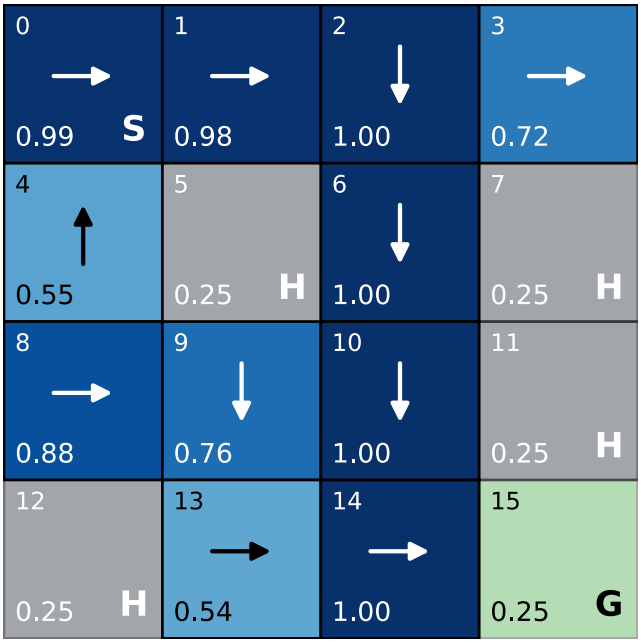
We train twelve agents from the uniform policy, each with its own seed, and plot the mean return of each training batch, an estimate of $J(\theta)$, against the ceiling $\gamma^5 \approx 0.774$ of a six-step path:

```
agents = [d2l.Policy.tabular(16, 4) for _ in range(12)]  # uniform policies
runs = np.array([list(train_reinforce_tabular(ac, env, gamma, seed))
                 for seed, ac in enumerate(agents)])     # (seed, update, 2)
curves, steps = runs[..., 0], runs[..., 1].astype(int)
d2l.plot_curves({'REINFORCE': curves}, xlabel='update',
                ylabel='mean return of the batch', reference=gamma ** 5)
```



The plot shows the median of the twelve runs, with a band from the slowest to the fastest. The median starts flat because, as the smallest case predicted, most early batches contain no success and give a zero estimate. Once a batch succeeds, its actions become more likely, successes become more frequent, and the mean return rises quickly. Every run approaches the dashed ceiling within a few dozen updates and stays a little below it, because the policy is still stochastic and sometimes leaves the shortest path. The next cell shows the learned policy of the first agent, with the most probable action and its probability at each cell:

```
probs = agents[0].probs(16, 4)
d2l.show_grid(env.unwrapped.desc, probs.max(axis=1), probs.argmax(axis=1))
```



The policy concentrates on one of the three shortest paths to the goal, chosen by the episodes that happened to succeed first. The holes and the goal stay uniform, because no action is ever taken there. Cells off the chosen path keep most of the preferences they picked up from early successful episodes that passed through them. They change only when an episode strays through them, which becomes rare as the path firms up; between visits Adam's momentum carries the last change on for a few dozen updates, then they stay put. Nothing in this loop evaluated the transition kernel: `rollout` recorded the next states that the simulator sampled, and the update read only states, actions and returns. This is the cancellation of Section 14.6.2.4 at work, which the verifier task, having no kernel, could not show.

14.6.4 The Policy Gradient Theorem

REINFORCE weights the score of every action by the return of the whole trajectory, including rewards that arrived before the action was taken. An action cannot affect rewards that are already collected, so those rewards should contribute nothing on average. The policy gradient theorem (Sutton et al. 2000) makes this precise, and it rewrites the gradient as an average over the states the policy visits rather than over whole trajectories. To state it, let the chain stay in its terminal state after the episode ends, with zero reward, so that s_t is defined for every t. Write $d_t^{\pi_\theta}(s) = \Pr(s_t = s)$ for the distribution of s_t under π_θ, and

$$d_\gamma^{\pi_\theta}(s) = (1-\gamma) \sum_{t \geq 0} \gamma^t \, d_t^{\pi_\theta}(s) \tag{14.55}$$

for its discounted average over time, the *discounted state-occupancy distribution*. The factor $1-\gamma$ makes the weights γ^t sum to one, so $d_\gamma^{\pi_\theta}$ is a probability distribution over states.

14.6.4.1 Statement and Proof

Proposition (policy gradient theorem). *For the objective* $J(\theta) = \mathbb{E}_{s_0 \sim \mu_0}[V^{\pi_\theta}(s_0)]$,

$$\nabla_\theta J(\theta) = \mathbb{E}_{\tau \sim P(\cdot;\theta)}\Big[\sum_{t \geq 0} \gamma^t \, \nabla_\theta \log \pi_\theta(a_t \mid s_t)\, Q^{\pi_\theta}(s_t, a_t)\Big] = \frac{1}{1-\gamma}\, \mathbb{E}_{s \sim d_\gamma^{\pi_\theta},\, a \sim \pi_\theta(\cdot|s)}\Big[\nabla_\theta \log \pi_\theta(a \mid s$$

(14.56)

Proof. By Equation 14.53 and the cancellation, $\nabla_\theta J = \mathbb{E}\big[\sum_t \nabla_\theta \log \pi_\theta(a_t \mid s_t)\, R(\tau)\big]$. At each t, split the return into the rewards before t and those from t on. The rewards before t are fixed once the agent stands at s_t, while a_t is still to be drawn, and the score of a_t has mean zero given s_t: $\sum_a \pi_\theta(a \mid s) \nabla_\theta \log \pi_\theta(a \mid s) = \sum_a \nabla_\theta \pi_\theta(a \mid s) = \nabla_\theta 1 = 0$. So the earlier rewards contribute zero. For the rewards from t on, condition on (s_t, a_t): by the Markov property their expected discounted sum is $\gamma^t Q^{\pi_\theta}(s_t, a_t)$. This gives the first form. For the second, group the sum over t by the state visited: for any function f, $\mathbb{E}\big[\sum_t \gamma^t f(s_t, a_t)\big] = \sum_s \sum_t \gamma^t d_t^{\pi_\theta}(s)\, \mathbb{E}_{a \sim \pi_\theta(\cdot|s)} f(s, a)$, and substitute the definition of $d_\gamma^{\pi_\theta}$. At terminal states $Q^{\pi_\theta} = 0$, so the steps after the end of an episode contribute nothing. ■

14.6.4.2 Consequences

The theorem has three consequences. First, the score at time t needs only the rewards from t on, because the proof removed the earlier ones as zero-mean noise, so an estimator that drops them has the same mean and usually less noise. Second, the gradient is an average over the states the policy visits, weighted by how often and how early it visits them, so a state the policy rarely reaches receives little gradient, and a batch that never visits it gives its row an estimate of exactly zero; this is why the cells off the learned path above barely change. Third, many implementations drop the factor γ^t from the first form. This replaces $d_\gamma^{\pi_\theta}$ by the undiscounted visitation of states, changes the relative weight of states visited late, and in general yields an update that is not the gradient of any objective (Nota and Thomas 2020). Exercise 8 examines the difference.

14.6.5 Costs and Limitations

14.6.5.1 The On-Policy Data Requirement

The estimator Equation 14.54 averages trajectories drawn from the *current* policy. After one update the parameters have changed, and the trajectories already collected no longer follow $P(\tau; \theta)$. Methods with this requirement are called *on-policy*, and they collect a new batch for every update. Q-learning is *off-policy*: its update learns about the greedy policy from transitions that any behavior policy generated (Section 14.3). We count the data that REINFORCE uses in environment steps:

```
hit = (curves >= 0.7).argmax(axis=1) + 1   # batches until the mean is 0.7
to_hit = np.array([s[:k].sum() for s, k in zip(steps, hit)])
for name, x in [('batches until a batch mean reaches 0.7', hit),
                ('environment steps until then', to_hit),
                ('environment steps of the full run', steps.sum(axis=1))]:
    print(f'{name:>38}: median {np.median(x):>6,.0f}, '
```

```
            f'range {x.min():,} to {x.max():,}')
```

```
batches until a batch mean reaches 0.7: median     32, range 25 to 49
         environment steps until then: median  4,313, range 3,210 to 6,191
     environment steps of the full run: median 27,049, range 26,058 to 28,383
```

A batch mean first reaches 0.7, about 90% of the ceiling, after 25 to 50 updates, which is 3,000 to 7,000 environment steps. The full run of 256 updates uses about 27,000 steps, because each update keeps collecting a new batch long after the policy has stopped improving. On the same map, tabular Q-learning with $\alpha = 0.9$ found, in almost every run, a greedy table that reaches the goal, and the whole run took about 3,500 steps (Section 14.3). Reusing a batch for several updates is the obvious remedy, and it is unsound without a correction (exercise 5).

14.6.5.2 Unbiasedness and Variance

The lake, like the verifier task, permits a check against the exact gradient, because its model is known. In Section 14.2 the value of a policy solved the linear system $V^\pi = r_\pi + \gamma P_\pi V^\pi$, where $P_\pi(s' \mid s) = \sum_a \pi(a \mid s) P(s' \mid s, a)$ and $r_\pi(s) = \sum_a \pi(a \mid s)\, r(s, a)$. The objective is therefore

$$J(\theta) = V^{\pi_\theta}(s_0) = \left[\left(I - \gamma P_{\pi_\theta} \right)^{-1} r_{\pi_\theta} \right]_{s_0}. \tag{14.57}$$

Every entry of P_{π_θ} and r_{π_θ} is a smooth function of θ, so automatic differentiation through the linear solve gives the exact $\nabla_\theta J$. Holes and the goal are absorbing states with zero reward in `TabularMDP.from_gym`, so the solve accounts for termination. We freeze a policy `probe` after 16 updates of the run with seed 3, which has started to climb by then:

```
probe = d2l.Policy.tabular(16, 4)
list(train_reinforce_tabular(probe, env, gamma, 3, num_updates=16))  # train
mdp = d2l.TabularMDP.from_gym(env, gamma)
```

`policy_objective` computes $J(\theta)$ by Equation 14.57 from a preference table, a kernel `P[s, a, s']` and rewards `r[s, a]`, and the framework's `grad` differentiates it through the solve. We compute J and its exact gradient at the probe:

```
def policy_objective(theta, P, r, gamma):
    """J(theta) = V(s0) of the softmax policy, a differentiable linear solve."""
    pi = torch.softmax(theta, -1)
    P_pi = torch.einsum('sa,sat->st', pi, P)
    return torch.linalg.solve(torch.eye(len(P)) - gamma * P_pi,
                              (pi * r).sum(-1))[0]

P, r = torch.as_tensor(mdp.P).float(), torch.as_tensor(mdp.r).float()
theta = probe.policy.weight.detach()
g_exact = torch.func.grad(policy_objective)(theta, P, r, gamma).numpy()
print(f'J(theta) after 16 updates = '
      f'{float(policy_objective(theta, P, r, gamma)):.3f}, '
      f'against the ceiling gamma^5 = {gamma ** 5:.3f}')
```

```
J(theta) after 16 updates = 0.313, against the ceiling gamma^5 = 0.774
```

The probe lies partway between the uniform policy and the ceiling. Now hold θ fixed and draw the estimator many times. `tabular_score` sums the weighted scores $w_t\,(e_{a_t} - \pi_\theta(\cdot \mid s_t))$ of a batch, each into the row of its state, by Equation 14.49. `reinforce_estimate` computes one draw of Equation 14.54 with it, as `verifier_step` does: each step is weighted by its episode's return, and the sum is divided by n. For each batch size n we draw fifty estimates and print three numbers relative to $\|\nabla_\theta J\|$: the root-mean-square error of one draw, the error of the mean of the fifty, and the error that an unbiased estimator would leave in that mean, the first number divided by $\sqrt{50}$:

```
def tabular_score(obs, act, w, pi):
    """sum_t w_t (e_{a_t} - pi(.|s_t)), each term added into row s_t: the
    weighted softmax score (eq_softmax_score) of a batch, the shape of pi."""
    u = np.zeros(pi.shape)
    np.add.at(u, obs, w[:, None] * (np.eye(pi.shape[1])[act] - pi[obs]))
    return u

def reinforce_estimate(env, probe, gamma, n, rng):
    """One draw of eq_reinforce at the frozen tabular policy `probe`: each
    step adds its trajectory's return times its score to row s; the sum is
    divided by n."""
    pi = probe.probs(env.observation_space.n, env.action_space.n)
    batch = rollout(env, probe.act, n, rng)
    w = batch.per_step(batch.episode_returns(gamma))
    return tabular_score(batch.obs, batch.act, w, pi).ravel() / n

rng, g = np.random.default_rng(4), g_exact.ravel()
for n in [4, 16, 64]:
    U = np.stack([reinforce_estimate(env, probe, gamma, n, rng)
                  for _ in range(50)])
    zero = int((U == 0).all(axis=1).sum())
    one = np.sqrt(((U - g) ** 2).sum(axis=1).mean()) / np.linalg.norm(g)
    mean = np.linalg.norm(U.mean(axis=0) - g) / np.linalg.norm(g)
    print(f'n={n:>2}: {zero:>2}/50 zero; one draw off by {one:.2f}, the mean '
          f'of 50 by {mean:.2f} (if unbiased: {one / np.sqrt(50):.2f})')
```

```
n= 4:  3/50 zero; one draw off by 3.85, the mean of 50 by 0.48 (if unbiased: 0.54)
n=16:  0/50 zero; one draw off by 1.78, the mean of 50 by 0.24 (if unbiased: 0.25)
n=64:  0/50 zero; one draw off by 0.96, the mean of 50 by 0.10 (if unbiased: 0.14)
```

At $n = 4$ a single draw is off by more than three times the length of the gradient it estimates, and some draws can be zero because no episode in the batch reached the goal. The error halves each time n quadruples, the $1/\sqrt{n}$ law of the smallest case. At every n the mean of fifty draws is off by about as much as an unbiased estimator would be, so the test finds no bias; exercise 6 pushes it to many more draws, where a small bias would show.

14.6.5.3 Plateaus and Convergence

The objective is not concave in θ. Already in the smallest case $J = p$ is a logistic function of $\theta_1 - \theta_2$, flat on both sides. Gradient ascent on a nonconcave function can in general stop at a poor stationary point. For the tabular softmax policy it cannot: there is no suboptimal stationary point, and exact gradient ascent with a small enough step converges to an optimal

policy when the start distribution gives every state positive probability (A. Agarwal et al. 2021; Mei et al. 2020). Convergence can still be slow. Where the better action at a state has small probability, the gradient of its preference carries that probability as a factor, so ascent creeps across a plateau, and there are MDPs in which exact softmax ascent needs a number of steps exponential in the number of states (Li et al. 2021).

Sampled updates with a fixed step size add a second risk. A few lucky batches can make the policy at some state nearly deterministic on a worse action, after which the gradient that would undo the choice is tiny, and the run stays committed for a long time (Mei et al. 2021). On the calm lake all twelve runs ended within a few hundredths of the ceiling; on slippery ice the same recipe settles at different levels from seed to seed, some well below the optimum, and exercise 7 finds out why.

We can now improve a policy from its own episodes and their rewards alone, without a model, an expert, or a table of values, and on the verifier task the same update already has the shape of post-training a language model: a response is a trajectory, and its reward arrives at the end. The estimate is right on average, but at the probe policy, with four episodes per batch, a single draw is off by more than three times the gradient's length, and every failed episode contributes nothing. Section 14.7 asks how much of that noise can be removed without moving the mean.

14.6.6 Summary

Policy gradient methods improve a parameterized policy π_θ directly by gradient ascent on its expected return $J(\theta)$. The log-derivative trick writes $\nabla_\theta J$ as an average over the policy's own trajectories of the return times the trajectory's score, and the transition kernel drops out of the score, so the gradient can be estimated from sampled episodes without a model. REINFORCE weights the score of every action by the return of its episode, which is behavior cloning's update with returns in place of unit weights. On the verifier task and the calm lake the estimate matches the exact gradient on average, but on the lake a small batch is off by several times the gradient's length, and the error shrinks only as $1/\sqrt{n}$. The policy gradient theorem shows that rewards collected before an action contribute only noise; every update needs a fresh batch from the current policy.

14.6.7 Exercises

1. **Parameter-dependent dynamics.** The derivation of Section 14.6.2.4 assumes that the transition probabilities do not depend on θ. A recommender violates this assumption when users who see an item more often click it less.
    1. Suppose the dynamics are $P_\theta(s_{t+1} \mid s_t, a_t)$. Repeat the derivation of $\nabla_\theta \log P(\tau;\theta)$ and state the term that no longer vanishes. What would the agent need to know to compute it?
    2. In a one-step model, the recommender shows item 1 with probability $\pi_\theta(1) = \sigma(\theta)$ and item 0 otherwise, where σ is the logistic function. A shown item 1 is clicked with probability $1-\sigma(\theta)$, item 0 is never clicked, and a click earns reward 1. Compute $J(\theta)$ and $\mathrm{d}J/\mathrm{d}\theta$. Compare the derivative with the expectation of the REINFORCE estimate Equation 14.54, which drops that term. Where does plain gradient ascent converge with each of the two?
2. code **Sparse rewards.** The smallest case of the text has one state, two actions, reward 1 for action 1 and 0 for action 2, and $p = \pi_\theta(1)$; a single-sample estimate has stan-

dard deviation $\sqrt{(1-p)/p}$ times its mean. On the lake the analogous quantity is the probability $p_0 \approx 0.014$ that an episode of the uniform policy reaches the goal, which Section 14.2.4 computes by policy evaluation at $\gamma = 1$.

1. A batch with no success gives the estimate zero, and `policy_step` then leaves the zero table unchanged, so the policy stays uniform until the first success. What distribution does the number of updates up to and including the first nonzero estimate follow, and what is its mean for batches of 16 episodes? Record that number for each of the twelve training runs and compare it with the prediction and with the flat start of the training curve.
2. Show that the mean of n independent single-sample estimates has standard deviation $\sqrt{(1-p)/(np)}$ times its mean. How many episodes per batch would make the standard deviation equal to the mean at $p = p_0$? How does the answer change as training raises the success probability?

3. **Shifted rewards.** A trajectory with return zero drops out of Equation 14.54, so subtracting one from every reward is a tempting way to make failed episodes informative.
   1. In the two-action setting of the smallest case, change the rewards to 0 for action 1 and -1 for action 2, which subtracts one from every return. Show that the mean of the single-sample estimate is unchanged and that the ratio of its standard deviation to its mean becomes $\sqrt{p/(1-p)}$. For which policies does the shift help, and for which does it hurt?
   2. On FrozenLake, the same subtraction makes every step earn -1 and the step into the goal earn 0. Show that the return of an episode of T steps decreases by $(1-\gamma^T)/(1-\gamma)$. Does this shift change J by a constant, as in the two-action setting?
   3. The shift is the per-step charge of problem 11b of Section 14.2, taken to $c = -1$. At $\gamma = 0.95$, compare the shifted return of the shortest path from the start into a hole with that of a shortest path to the goal on the calm lake. Which policy maximizes J under the shifted rewards? Among the failed episodes of a batch, which ones does the estimate Equation 14.54 penalize least?

4. code **The theorem in a table.** ◻ For the tabular softmax policy, write $\rho_\gamma(s) = \sum_{t\geq 0} \gamma^t d_t^{\pi_\theta}(s)$ for the discounted number of visits to s, so that $d_\gamma^{\pi_\theta} = (1-\gamma)\rho_\gamma$ in Equation 14.56.
   1. Combine the occupancy form of Equation 14.56 with Equation 14.49 to show that
   $$\frac{\partial J}{\partial \theta_{s,a'}} = \rho_\gamma(s)\, \pi_\theta(a' \mid s)\big(Q^{\pi_\theta}(s,a') - V^{\pi_\theta}(s)\big).$$
   2. By problem 1b of Section 14.2, $\rho_\gamma^\top$ is the row of $(I - \gamma P_{\pi_\theta})^{-1}$ that belongs to s_0. Compute ρ_γ, V^{π_θ} and Q^{π_θ} at the parameters of `probe` by linear solves with `mdp.P` and `mdp.r`, assemble the formula, and compare the result with `g_exact`.
   3. On slippery ice a command a moves the agent in direction a or in either perpendicular direction with probability $1/3$ each. Show that at the uniform policy both maps induce the same chain, hence the same ρ_γ and V, and that the slippery $Q(s,a)$ satisfies $Q_{\text{slip}}(s,a) - V(s) = -\frac{1}{3}\big(Q_{\text{calm}}(s,\bar{a}) - V(s)\big)$, where $\bar{a}$ is the command opposite to a. Conclude that the exact gradient at $\theta = 0$ is one third as long on slippery ice, and check it with `J_fn` on both transition tables.

5. code **Reusing a batch.** ◻ Modify `train_reinforce_tabular` to call `policy_step` k times on each batch without any correction, for $k \in \{1, 2, 5, 20\}$. Write θ_{old} for the parameters that collected the batch.
    1. On the calm lake, log the batch mean return and, after the k-th step, the mean and the maximum over the batch of the ratio $\pi_\theta(a_t \mid s_t)/\pi_{\theta_{\text{old}}}(a_t \mid s_t)$; `log_prob_np` evaluates both policies. How many environment steps does each k need before the batch mean return reaches 0.7, and how large does the ratio grow?
    2. As k grows, the steps approach a maximizer of the surrogate $\sum_t w_t \log \pi_\theta(a_t \mid s_t)$ on the fixed batch, where w_t is the return of the episode that contains step t. Find the maximizing tabular policy in closed form, and use it to explain the result of part a.
    3. Compare $k = 1$ with $k = 20$ on the slippery map (`is_slippery=True`) by the final $J(\theta)$, computed with Equation 14.57 from the slippery transition table. Each run takes seconds. Explain the difference from the calm lake with the closed form of part b.
6. code **Testing unbiasedness.** ◻ The fifty-draw test of the text cannot reveal a bias smaller than the noise of fifty draws. A sharper test averages M estimates: for an unbiased estimator, the error of the average keeps shrinking as M grows, whereas for a biased one it levels off at the bias.
    1. At the parameters of `probe`, draw $M = 25{,}600$ estimates with `reinforce_estimate` at $n = 1$. Plot the error of the running average relative to $\|$`g_exact`$\|$ against M on logarithmic axes. Explain the slope of the curve.
    2. Repeat the measurement with an environment rebuilt with `max_episode_steps` lowered to 20 and to 10, passed to `reinforce_estimate`, so that `rollout` ends some episodes by truncation. At what level does each curve settle? Of which objective are the truncated estimates unbiased estimates of the gradient?
7. **Softmax plateaus.** ◻ Consider a single state with two actions, rewards 1 and 0, and initial preferences $\theta = (0, L)$ with $L > 0$, so that the better action starts with probability $1/(1 + e^L)$. This is the plateau of Section 14.6.5.3 in its smallest form.
    1. Show that J depends on θ only through $x = \theta_1 - \theta_2$, as $J = \sigma(x)$ with the logistic function σ. Conclude that J is not concave.
    2. Show that one step of plain gradient ascent with step size η increases x by $2\eta\,\sigma(x)(1 - \sigma(x))$. Treat this as a differential equation, and show that reaching $\pi_\theta(1) = 1/2$ from $x = -L$ takes about $e^L/(2\eta)$ steps when L is large.
    3. Adam, fed the same exact gradients, moves each coordinate by about its learning rate η per step while the sign of the gradient stays fixed. Estimate the number of steps it needs, and compare it with the answer of part b at $L = 10$ and $\eta = 0.1$. What does the comparison suggest about the use of Adam in `Policy.tabular`?
    4. code Run `train_reinforce_tabular` for 1,024 updates on the slippery map with several seeds and compute the final $J(\theta)$ with Equation 14.57. At which states is the final policy of a run that ends well below $V^*(s_0)$ nearly deterministic, and on which actions? Compute $\|\nabla_\theta J\|$ there and at the uniform start. Is the final θ a stationary point? Run exact gradient ascent from it and from $\theta = 0$, and relate what you see to parts b and c.
8. code **The dropped discount.** ◻◻ Many implementations omit the factor γ^t from the trajectory form of Equation 14.56, which gives the update field

$$F(\theta) = \mathbb{E}_{\tau \sim P(\cdot;\theta)}\Big[\sum_{t \geq 0} \nabla_\theta \log \pi_\theta(a_t \mid s_t)\, Q^{\pi_\theta}(s_t, a_t)\Big]. \tag{14.58}$$

Nota and Thomas (2020) prove that F need not be the gradient of any function, and they construct an MDP in which exact updates along F drive the policy toward the worst possible policy. The parts below check the first finding on FrozenLake and then construct a small MDP in which a step along F lowers J, the starting point of the second.

1. Following problem 4a, show that

$$F_{s,a'}(\theta) = \rho_1(s)\, \pi_\theta(a' \mid s)\big(Q^{\pi_\theta}(s, a') - V^{\pi_\theta}(s)\big), \tag{14.59}$$

   where $\rho_1(s) = \sum_{t \geq 0} \Pr(s_t = s,\ t < T)$ is the expected number of visits to s before the episode ends. Compute ρ_1 by a linear solve in which the rows of the policy's transition matrix that belong to terminal states are set to zero.

2. The Jacobian of a continuously differentiable gradient field is a Hessian and therefore symmetric. Compute the Jacobian DF at the zero table and at the parameters of `probe` by automatic differentiation, and report $\|DF - DF^\top\| / \|DF\|$ in the Frobenius norm. Run the same check on $\nabla_\theta J$ as a control.

3. Measure the cosine between F and $\nabla_\theta J$ at the same two points. Give a practical reason why implementations accept F anyway.

4. Show that $F(\theta)^\top \nabla_\theta J(\theta) \geq 0$ for the tabular softmax policy at every θ, so that a small step along F never lowers J on FrozenLake. Then construct a small episodic MDP in which a single parameter controls the policy at two states visited in sequence, together with a θ at which $F(\theta)\, \mathrm{d}J/\mathrm{d}\theta < 0$.

9. **The Gaussian step of the figure.** The policy of Figure 14.13 is $\mathcal{N}(a; \mu, 1)$ and the reward is $R(a) = 0.4 + 2e^{-(a-2)^2/2}$.

   1. Show that $J(\mu) = 0.4 + \sqrt{2}\, e^{-(\mu-2)^2/4}$, and check the values $J = 0.92$ and $\mathrm{d}J/\mathrm{d}\mu = 0.52$ at $\mu = 0$.
   2. Show that the score with respect to μ is $a - \mu$ and verify directly that $\mathbb{E}[R(a)(a - \mu)] = \mathrm{d}J/\mathrm{d}\mu$. Compute the variance of the single-sample estimate $R(a)(a - \mu)$ at $\mu = 0$, and the standard deviation of the twelve-sample estimate relative to its mean. Is the figure's step to 0.32 unusual?
   3. code In NumPy, run the twelve-sample update with $\eta = 1$ for 50 steps from $\mu = 0$ with 100 seeds. Where does μ settle, and how does its spread around that point depend on η?

14.7 Baselines and Variance Reduction

In the previous section we trained a policy by raising the log-probability of every sampled action in proportion to the return of its trajectory,

$$\hat{u} = \frac{1}{n} \sum_{i=1}^{n} R(\tau_i) \sum_{t=0}^{T-1} \nabla_\theta \log \pi_\theta(a_t^i \mid s_t^i). \tag{14.60}$$

On average this estimate is the gradient of the expected return $J(\theta)$. In a single draw it is not: with four episodes per update, a single estimate at a partly trained policy was typically off by more than three times the gradient's length, and a batch without a success gave no update at all. The weights explain much of this. On the calm lake every return is either zero or γ^{T-1}, so an action is either ignored or rewarded. A detour on a successful episode is pushed up, and the step into a hole on a failed one is never pushed down.

Williams saw the problem in the 1992 paper that introduced REINFORCE and proposed a remedy there: subtract a reference level, a *baseline*, from the return before weighting (Williams 1992). Statisticians call this a control variate. Subtract something whose mean you know, and the estimate keeps its mean while its variance can fall. Starting from one fact, that the score of the policy's own action averages to zero, this section works out which quantities may be subtracted and how much variance each removes, measured against exact gradients. The actor-critic methods of the next chapter and the group-based updates used to post-train language models are this estimator with other baselines.

```
import gymnasium as gym
```

We keep the calm lake, $\gamma = 0.95$ and the softmax table of Section 14.6. On the calm lake every action visibly affects the return; on slippery ice success is mostly luck, the return says little about any one action (at the uniform start, Section 14.6 found, a batch must be about nine times larger there for the same precision). Each update uses four episodes, few enough for differences in variance to show, and each training comparison runs twenty seeds, a hundred at the largest step size. Episodes are cut after 200 steps. Only a policy stuck in a loop walks that long, and its value is near zero, so the cut changes no weight measurably; without it, the large step sizes used below produce episodes thousands of steps long.

```
gamma, eta_v = 0.95, 0.1              # discount; value-table step size
num_updates, batch_episodes = 150, 4  # small batches, to expose variance
num_seeds = 20                        # twenty seeds per arm
num_seeds_large = 100                 # at the large step size, eta = 16
env = gym.make('FrozenLake-v1', is_slippery=False, max_episode_steps=200)
```

14.7.1 The Zero-Mean Identity

Every method in this section changes the weight of a score and must leave the mean in place. One identity, already used in the proof of Equation 14.56, guarantees this; we state it here for every differentiable policy.

14.7.1.1 The Zero-Mean Lemma

For the softmax table, the score of action a at state s has one nonzero block, at the preferences of state s:

$$\frac{\partial \log \pi_\theta(a \mid s)}{\partial \theta_{s,a'}} = \mathbf{1}(a' = a) - \pi_\theta(a' \mid s). \tag{14.61}$$

Averaged over the policy's own action at s, the indicator averages to $\pi_\theta(a' \mid s)$ and cancels the second term. The cancellation holds for every differentiable policy.

Lemma. For every state s, $\sum_{a \in \mathcal{A}} \pi_\theta(a \mid s)\, \nabla_\theta \log \pi_\theta(a \mid s) = 0$.

Proof. $\sum_a \pi_\theta(a \mid s) \nabla_\theta \log \pi_\theta(a \mid s) = \sum_a \nabla_\theta \pi_\theta(a \mid s) = \nabla_\theta \sum_a \pi_\theta(a \mid s) = \nabla_\theta 1 = 0$. ■

In words: at any state, the score of the sampled action has mean zero. Probability mass that one action gains, the others lose.

14.7.1.2 The Conditional Identity

The useful consequence is slightly stronger. Suppose c is any quantity that is already determined by the time the agent stands at state s_t: a reward collected earlier in the trajectory, a constant, or a function of s_t itself. Conditioned on the trajectory prefix $(s_0, a_0, r_0, \dots, s_t)$, the value c is fixed while the action a_t is still random, and since the policy consults only s_t, the inner expectation is the one the lemma covers:

$$\mathbb{E}\big[c\, \nabla_\theta \log \pi_\theta(a_t \mid s_t)\big] = \mathbb{E}\Big[c\, \underbrace{\mathbb{E}\big[\nabla_\theta \log \pi_\theta(a_t \mid s_t) \mid s_0, a_0, \dots, s_t\big]}_{=\,0 \text{ by the lemma}}\Big] = 0. \tag{14.62}$$

We can therefore multiply any score in the REINFORCE estimator by such a quantity, or subtract such a quantity from its weight, without moving the average. The condition is about the prefix: a quantity that depends on a_t or on anything that happens after it is not covered.

14.7.2 Reward-to-Go, Baselines, and Control Variates

The identity licenses two operations on the weight at step t: dropping terms fixed before a_t, and subtracting a quantity fixed before a_t. Neither moves the mean. Whether either reduces the variance is a separate question.

14.7.2.1 Reward-to-Go and Causality

In the REINFORCE estimator, the score at step t is multiplied by the return $R(\tau_i)$ of the whole trajectory, including rewards collected before action a_t^i was drawn. Those rewards are fixed by the prefix, so by the conditional identity their product with the score at step t has mean zero. Removing them leaves

$$\nabla_\theta J(\theta) = \mathbb{E}_{\tau \sim P(\cdot;\theta)}\Big[\sum_{t=0}^{T-1} \nabla_\theta \log \pi_\theta(a_t \mid s_t) \sum_{t'=t}^{T-1} \gamma^{t'} r_{t'}\Big]. \tag{14.63}$$

This is *causality*: an action can affect the future, not the past. The inner sum equals $\gamma^t \hat{G}_t$, where

$$\hat{G}_t = \sum_{t'=t}^{T-1} \gamma^{t'-t}\, r_{t'} \tag{14.64}$$

is the *reward-to-go* from step t, the discounted return of the rest of the episode. The plain estimator multiplies a sum of T scores by a return that is itself a sum over T steps, so its variance grows quickly with the length of the episode when rewards arrive at every step;

removing the earlier rewards shortens the sum that multiplies each score. Removing a term of mean zero does not by itself lower the variance, though, because the removed term may be correlated with what remains; the measurements below show both outcomes.

Implementations, ours included, also drop the factor γ^t and weight the score at step t by $\hat{G}_t$ alone:

$$\hat{u} = \frac{1}{n} \sum_{i=1}^{n} \sum_{t=0}^{T-1} \hat{G}_t^i \, \nabla_\theta \log \pi_\theta(a_t^i \mid s_t^i). \tag{14.65}$$

Dropping γ^t is not licensed by the identity. It counts a late visit to a state as heavily as an early one, so for $\gamma < 1$ the mean of Equation 14.65 tilts away from $\nabla_\theta J$. Thomas (2014) identified this bias in actor-critic algorithms, and Nota and Thomas (2020) showed that the update is the gradient of no objective at all. At the policy studied below the tilt is about eight degrees (a cosine of 0.991) and the mean is about 1.4 times longer (exercise 11). We keep the convention, which the methods built on this estimator share, and measure both versions.

The reward-to-go of every step follows from one backward pass per episode, $\hat{G}_t = r_t + \gamma \hat{G}_{t+1}$, starting from zero after the last step. `backward_scan` runs this recursion for any vector and factor (later sections scan other quantities with other factors) over each episode that `Batch.episodes` of Section 14.6 reports, and `reward_to_go` applies it to the rewards with factor γ:

```
@d2l.add_to_class(d2l.Batch)
def backward_scan(self, x, factor):
    """y_t = x_t + factor * y_{t+1}, restarted at every episode boundary."""
    y = np.zeros_like(x)
    for ep in self.episodes():
        running = 0.0
        for t in reversed(range(ep.start, ep.stop)):
            running = x[t] + factor * running
            y[t] = running
    return y

@d2l.add_to_class(d2l.Batch)
def reward_to_go(self, gamma):
    """G_t: the discounted return of the rest of its episode, by one scan."""
    return self.backward_scan(self.rew, gamma)
```

14.7.2.2 Baselines

The second operation is subtraction. Any quantity b that is determined by the trajectory prefix up to s_t, whether a constant or a function $b(s_t)$ of the current state, can be subtracted from the reward-to-go:

$$\hat{u} = \frac{1}{n} \sum_{i=1}^{n} \sum_{t=0}^{T-1} \left(\hat{G}_t^i - b(s_t^i)\right) \nabla_\theta \log \pi_\theta(a_t^i \mid s_t^i). \tag{14.66}$$

Such a b is called a *baseline*. The extra term is $b(s_t)$ times the score, which has mean zero by the conditional identity, so Equation 14.66 and Equation 14.65 have the same mean.

On the lake no reward-to-go is negative, so without a baseline no sampled action is ever pushed down. Subtracting a value near the typical reward-to-go gives positive weights to continuations that did better than typical and negative weights to those that did worse. If the mean is unchanged, how can learning change? Through the variance, which the next subsection computes. And why not subtract an estimate of $Q(s_t, a_t)$, the best predictor of the weight? It depends on a_t, so the identity no longer applies and the mean moves (exercise 1).

14.7.2.3 Control Variates

Baseline subtraction is an instance of a general method for Monte Carlo estimation. Suppose we want $\mathbb{E}[X]$, and each sample also contains a correlated quantity Y whose expectation is known. For any scalar c, define

$$X_c = X - c\,(Y - \mathbb{E}[Y]). \tag{14.67}$$

Since $\mathbb{E}[Y - \mathbb{E}[Y]] = 0$, the estimator X_c has the same expectation as X for every c. Its variance is

$$\operatorname{Var}(X_c) = \operatorname{Var}(X) - 2c \operatorname{Cov}(X, Y) + c^2 \operatorname{Var}(Y),$$

a quadratic in c, minimized at

$$c^* = \frac{\operatorname{Cov}(X, Y)}{\operatorname{Var}(Y)}. \tag{14.68}$$

Substituting c^* back in leaves the variance at $(1 - \operatorname{corr}^2) \operatorname{Var}(X)$, where corr is the correlation between X and Y. Strong correlation permits a large reduction: at $\operatorname{corr} = 0.9$ the variance falls by a factor of about five, whereas at $\operatorname{corr} = 0$ it is unchanged. The quantity Y is called a *control variate* for X.

For a policy gradient, take $X = \hat{G}_t \nabla_\theta \log \pi_\theta(a_t \mid s_t)$ and $Y = \nabla_\theta \log \pi_\theta(a_t \mid s_t)$. By the lemma $\mathbb{E}[Y \mid s_t] = 0$, and $c = b(s_t)$ recovers Equation 14.66: a baseline is a control variate built from the score. The best coefficient is c^*, computed separately for each state; for a vector-valued score, inner products replace the products and give the optimal baseline b^* of exercise 3. Greensmith et al. (2004) develop this analysis for policy gradients.

The smallest case shows the size of the effect. Take the one-step problem of Figure 14.13: one action a drawn from the normal distribution $\mathcal{N}(a; \mu, 1)$ at $\mu = 0$, and the reward $R(a) = 0.4 + 2e^{-(a-2)^2/2}$. The expected reward is $J = \mathbb{E}[R] = 0.92$, the score with respect to μ is $a - \mu = a$, and the exact gradient is $\mathrm{d}J/\mathrm{d}\mu = 0.52$. A single-sample estimate $R(a)\,a$ has this mean and a standard deviation of 1.34, more than two and a half times the quantity it estimates. Subtracting the mean reward, $(R(a) - 0.92)\,a$, cuts the standard deviation to 0.69. The best constant is $b^* = \mathbb{E}[R(a)\,a^2]/\mathbb{E}[a^2] = 1.18$, above the mean reward because actions far from zero have large scores and, here, large rewards; it leaves $1 - \operatorname{corr}^2 = 0.23$ of the variance without a baseline (Figure 14.14).

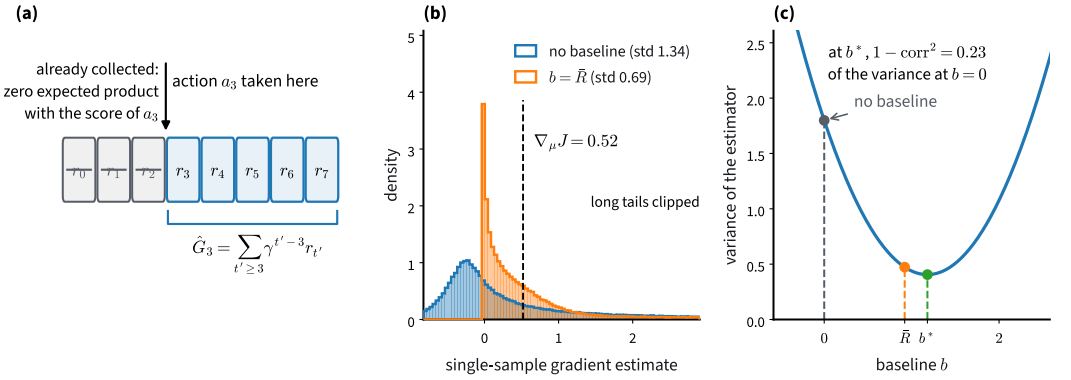


Figure 14.14: Two ways to quiet the score-function estimator. (a) In a multi-step episode, rewards collected before an action have zero expected product with its score, so only the reward-to-go $\hat{G}_3$ remains. (b) On the one-step problem of Figure 14.13, subtracting the mean reward cuts the standard deviation of a single-sample estimate from 1.34 to 0.69; both histograms are clipped at their 1st and 95th percentiles. (c) The variance as a function of a constant baseline b is a parabola, smallest at $b^* = 1.18$, where 0.23 of the variance at $b = 0$ remains.

14.7.2.4 The Advantage and the Learned Baseline

On the lake one constant cannot fit every state: a cell next to the goal has a much higher expected reward-to-go than the start. The natural state-dependent baseline is the value of the state. By Section 14.2, $V^\pi(s) = \sum_a \pi(a \mid s)\, Q^\pi(s,a)$, the policy's average over the values of its actions. The *advantage* of an action measures how much better it is than that average,

$$A^\pi(s,a) = Q^\pi(s,a) - V^\pi(s), \qquad \mathbb{E}_{a\sim\pi(\cdot|s)}\big[A^\pi(s,a)\big] = 0. \tag{14.69}$$

The mean is zero by the relation just stated. A positive advantage marks an action that beats the policy's average at s. Since $\mathbb{E}[\hat{G}_t \mid s_t, a_t] = Q^{\pi_\theta}(s_t,a_t)$, the weight $\hat{G}_t - V^{\pi_\theta}(s_t)$ is a sample of the advantage of the action taken, so the update raises the actions that beat the state's average and lowers the rest.

We do not know V^{π_θ}, but we can estimate it from the same batches. Keep a table $\hat{V}(s)$, and after each batch move the entry of every visited state toward the reward-to-go observed there,

$$\hat{V}(s_t) \leftarrow \hat{V}(s_t) + \eta_V\big(\hat{G}_t - \hat{V}(s_t)\big), \tag{14.70}$$

with a step size η_V. This is REINFORCE with a baseline (Williams 1992). The table is fitted by regression on Monte Carlo returns, the reward-to-go values of complete episodes. Targets built from $\hat{V}$'s own predictions one step later give the actor-critic methods of Chapter 15, whose advantage estimate reduces to $\hat{G}_t - \hat{V}(s_t)$ when it does not bootstrap at all.

14.7.3 Centering, Scaling, and Normalization

Practitioners often standardize the weights of each batch or compare the samples of a group with one another, and every implementation must decide what to divide a summed loss by. Each choice changes either the estimator or only its scale, and the two are easy to confuse. Telling them apart needs a fixed policy whose exact gradient we know.

We rebuild `probe` exactly as in Section 14.6, with `d2l.train_reinforce_tabular` and the same sixteen updates of sixteen episodes. Its value $J(\theta) = V^{\pi_\theta}(s_0)$ is the first entry of the solution of the linear system $V = r_\pi + \gamma P_\pi V$ of Section 14.2, with $P_\pi(s' \mid s) = \sum_a \pi_\theta(a \mid s)\, P(s' \mid s, a)$ and $r_\pi(s) = \sum_a \pi_\theta(a \mid s)\, r(s, a)$. `probe.probs` reads π_θ off the probe, and `mdp.policy_value(pi_np)` solves this system in NumPy and returns V^{π_θ} at every state, the exact state baseline:

```
mdp = d2l.TabularMDP.from_gym(env, gamma)
probe = d2l.Policy.tabular(16, 4)        # the probe of sec_policygradient
list(d2l.train_reinforce_tabular(probe, env, gamma, 3, num_updates=16))
pi_np = probe.probs(16, 4)
V_pi = mdp.policy_value(pi_np)           # the exact state baseline
```

`exact_gradient` differentiates `d2l.policy_objective` of Section 14.6, the same system written with the framework's arrays, at the probe's preference table. It takes the reward table `r` as an argument, so it also serves a modified reward later:

```
theta_probe, P = probe.policy.weight.detach(), torch.as_tensor(mdp.P).float()

def exact_gradient(r):
    """grad J at the probe for the rewards r[s, a], flattened."""
    return torch.func.grad(d2l.policy_objective)(
        theta_probe, P, torch.as_tensor(r).float(), gamma).numpy().ravel()

g_exact = exact_gradient(mdp.r)
print(f'J(theta) = {V_pi[0]:.3f}, |grad J| = {np.linalg.norm(g_exact):.3f}')
```

```
J(theta) = 0.313, |grad J| = 0.129
```

The printed $J(\theta) = 0.313$, against a best possible $\gamma^5 \approx 0.77$, is the probe's value in Section 14.6. Its exact gradient is the reference for every measurement below.

14.7.3.1 Batch Centering versus Variance Scaling

A common variant standardizes the reward-to-go values within each batch. Collect every $\hat{G}_t^i$ in the current batch, compute their mean $\bar{R}$ and standard deviation σ, and use

$$\tilde{G}_t^i = \frac{\hat{G}_t^i - \bar{R}}{\sigma + 10^{-8}} \tag{14.71}$$

in place of $\hat{G}_t^i$, where the constant 10^{-8} avoids dividing by zero. Subtracting $\bar{R}$ is a baseline: it changes the relative weights of the samples and can change the direction of the estimate. Because $\bar{R}$ comes from the same batch, it depends weakly on the sampled actions; the next

subsection computes the resulting bias for per-trajectory weights, and exercise 5 for the step-level weights used here.

Dividing by $\sigma + 10^{-8}$ multiplies the whole batch estimate by one positive number, so within a batch it acts as a step size that adapts to the spread of the rewards, not as a baseline. Across batches the factor is random and correlated with the batch, so the expected update tilts as well; exercise 6 works out the related effect of a σ that differs across prompts. On the lake $0 \le \hat{G}_t \le 1$ implies $\sigma \le 1/2$, so at a fixed step size normalization takes steps at least twice as long as centering; below we measure about three times. `standardize` implements Equation 14.71:

```
def standardize(x, eps=1e-8):
    """Center a NumPy batch of weights and divide by its standard deviation
    (NumPy's std, which divides by the batch size)."""
    return (x - x.mean()) / (x.std() + eps)
```

14.7.3.2 The Leave-One-Out Baseline

The batch mean includes the sample it is subtracted from. Leave-one-out centering removes this self-dependence by giving each trajectory the mean of the *other* trajectories as its baseline,

$$b_i = \frac{1}{n-1} \sum_{j \neq i} R(\tau_j).$$

Trajectories are drawn independently, so b_i is independent of trajectory i, and the conditional zero-mean identity applies with no correction: the estimator is unbiased at every batch size. Its weights are those of centering by $\bar{R}$, scaled,

$$R_i - b_i = \frac{n}{n-1}(R_i - \bar{R}), \tag{14.72}$$

so plain centering with per-trajectory weights multiplies the mean of the estimator by $(n-1)/n$, one half at $n = 2$. We verify both statements on a problem small enough to enumerate: two states visited in order, two actions at each, rewards from a table. `leave_one_out` computes the weights $R_i - b_i$ of one batch:

```
def leave_one_out(R):
    """R_i - b_i, with b_i the mean of the other n - 1 returns: n/(n-1)
    times the centered returns."""
    return (R - R.mean()) * len(R) / (len(R) - 1)
```

The next cell computes the exact gradient `g` from the four possible trajectories and the exact mean of both estimators over all sixteen ordered batches of two:

```
rng = np.random.default_rng(0)
th = rng.standard_normal((2, 2))              # a generic two-state table
pi2 = np.exp(th) / np.exp(th).sum(1, keepdims=True)
r2 = np.array([[0.3, 1.0], [0.6, 0.1]])       # r[s, a]; s0 -> s1 -> done
```

```
trajs = [(a0, a1) for a0 in range(2) for a1 in range(2)]
p = np.array([pi2[0, a0] * pi2[1, a1] for a0, a1 in trajs])
R = np.array([r2[0, a0] + r2[1, a1] for a0, a1 in trajs])
S = np.zeros((4, 2, 2))                          # the score of each trajectory
for i, (a0, a1) in enumerate(trajs):
    S[i] = -pi2
    S[i, 0, a0] += 1
    S[i, 1, a1] += 1
g = (p[:, None, None] * R[:, None, None] * S).sum(0)    # exact gradient
u_loo = u_cen = 0.0
for i in range(4):
    for j in range(4):                           # every batch of n = 2
        w, wc = leave_one_out(R[[i, j]]), R[[i, j]] - R[[i, j]].mean()
        u_loo += p[i] * p[j] * (w[0] * S[i] + w[1] * S[j]) / 2
        u_cen += p[i] * p[j] * (wc[0] * S[i] + wc[1] * S[j]) / 2
print(f'E[leave-one-out] equals the exact gradient: {np.allclose(u_loo, g)}')
print(f'E[centered] equals (n-1)/n of it:           {np.allclose(u_cen, g / 2)}')
```

```
E[leave-one-out] equals the exact gradient: True
E[centered] equals (n-1)/n of it:           True
```

Both identities hold to machine precision. Kool et al. (2019) proposed the leave-one-out baseline for REINFORCE with several samples per input, and under the name RLOO it is used to post-train language models (Ahmadian et al. 2024): sample n responses to a prompt and weight each by its reward minus the mean reward of the other $n-1$.

14.7.3.3 The Group as the Batch

On the lake every episode starts in the same cell, so one batch mean serves all of them. The verifier task of Section 14.1 is different: four arithmetic prompts, nine shared candidate responses, and a verifier that pays one for the correct answer; the reference policy succeeds with probability 0.062, and no policy can exceed 0.5 because two prompts have no correct candidate. Each prompt is its own start state with its own success rate, so the baseline must be per prompt. `verifier_step` of Section 14.6 samples K responses to each prompt and weights the score $\mathbf{e}_y - \pi(\cdot \mid x)$ of each response y by a function of the K rewards. The K responses to one prompt, the *group*, supply the baseline.

With rewards of zero and one, centering by the group mean has a simple form. If k of the K responses are correct, the group mean is k/K, so a correct response receives the weight $1 - k/K$ and an incorrect one $-k/K$. Without the baseline an incorrect response had weight zero; now it is pushed down, and harder the more often the policy already succeeds on that prompt. Dividing by the group's standard deviation $\sqrt{\hat{p}(1-\hat{p})}$, with $\hat{p} = k/K$, turns the two weights into

$$+\sqrt{\frac{1-\hat{p}}{\hat{p}}} \quad \text{and} \quad -\sqrt{\frac{\hat{p}}{1-\hat{p}}}. \tag{14.73}$$

With $K = 16$ and two correct answers, each correct response gets about $+2.65$ and each incorrect one about -0.38: a rare success on a hard prompt receives a large push. A group in which every response scores the same, all wrong or all right, has nothing to compare and gives every response the weight zero; Section 14.4 counted how often that happens.

The standardized weight of response j in its group is

$$\hat{A}_j = \frac{r_j - \bar{R}}{\sigma + 10^{-8}}, \tag{14.74}$$

where $\bar{R}$ and σ are the mean and standard deviation of the K rewards of the group. It is the per-trajectory form of Equation 14.71, with the prompt as the start state and the group as the batch: one reward per response is standardized, not one reward-to-go per step. It is the weight of Group Relative Policy Optimization (GRPO) (Shao et al. 2024), and `group_step` is `verifier_step` with `standardize` as the weighting:

```
def group_step(task, theta, K, reward, rng, lr=2.0):
    """verifier_step with the group-normalized weight (GRPO): the rewards
    of each prompt's K responses, standardized within the group."""
    return d2l.verifier_step(task, theta, K, reward, rng, standardize, lr)
```

The group mean includes the response's own reward, so by the leave-one-out calculation the expected group-centered update is $(K-1)/K$ times the gradient. We check this exactly for groups of two, without the division by σ. The cell computes the score of every response (`Sv`), the exact gradient of each prompt's success as the nine-term sum $\sum_y \pi(y \mid x)\, r(x,y)\, (\mathbf{e}_y - \pi(\cdot \mid x))$ (`gv`), and the exact mean of the centered update over all ordered pairs of responses (`u`):

```
task = d2l.VerifierTask()                  # the instance of sec_mdp
K, pi, Rv = 2, task.policy(task.theta_ref), task.R
Sv = np.eye(len(task.resp))[None, :, :] - pi[:, None, :]  # eq_softmax_score
gv = np.einsum('xy,xy,xyv->xv', pi, Rv, Sv)          # the exact gradient
u = np.zeros_like(gv)
for y1, y2 in np.ndindex(len(task.resp), len(task.resp)):  # groups of K = 2
    w = (Rv[:, y1] - Rv[:, y2]) / 2                  # reward minus group mean
    u += (pi[:, y1] * pi[:, y2] * w)[:, None] * (Sv[:, y1] - Sv[:, y2]) / K
print(f'E of the group-centered update is (K-1)/K of the gradient: '
      f'{np.allclose(u, gv * (K - 1) / K)}')
print(f'rescaled by K/(K-1), leave-one-out, it is exact: '
      f'{np.allclose(u * K / (K - 1), gv)}')
```

```
E of the group-centered update is (K-1)/K of the gradient: True
rescaled by K/(K-1), leave-one-out, it is exact: True
```

How much does the bias matter? At $K = 1$ the factor is zero: the group mean equals the response's own reward, every weight is zero, and the policy never moves, although its gradient is not zero. For $K \geq 2$ the factor only shrinks the step. The sweep fixes the budget at $6{,}400$ scored responses per prompt, so smaller groups take more steps ($3{,}200$ at $K = 2$, 200 at $K = 32$):

```
budget = 6400                    # scored responses per prompt, matched
for K in (1, 2, 4, 8, 32):
    rng, theta = np.random.default_rng(1), task.theta_ref.copy()
    for _ in range(budget // K):
```

```
        theta = group_step(task, theta, K, task.verify, rng)
    print(f'K = {K:2d}: success of a sampled response '
          f'{task.success(task.policy(theta)):.3f}')
```

```
K =  1: success of a sampled response 0.062
K =  2: success of a sampled response 0.500
K =  4: success of a sampled response 0.500
K =  8: success of a sampled response 0.500
K = 32: success of a sampled response 0.500
```

At $K = 1$ the policy keeps the reference score 0.062, as predicted, and every $K \geq 2$ reaches the ceiling of 0.5 within the same budget, for each of the ten seeds we tried. The group supplies the per-prompt comparison that a learned value table would otherwise supply. Dividing by σ per prompt has a further effect that this instance does not show: it changes which prompts the update favors (exercise 6).

14.7.3.4 The Loss Divisor

The double sum in Equation 14.65 runs over episodes and steps, and an implementation must divide it by something. Dividing by the number of episodes n gives that equation. Dividing by the total number of steps rescales each batch by the reciprocal of its mean episode length, which changes as the policy improves. Dividing each episode by its own length reweights the episodes and can change the direction. A fixed constant changes only the scale. We compute all four on one batch of four episodes of different lengths; `agg` sums the scores of batch `b` under the policy `pi` with `d2l.tabular_score` of Section 14.6, each weighted by its reward-to-go times a per-step factor `scale`:

```
b = d2l.rollout(env, probe.act, 4, np.random.default_rng(6))
G, T = b.reward_to_go(gamma), np.diff(b.ep_ends, prepend=0)
print(f'episode lengths {T}, successes {int(b.rew.sum())}')

def agg(b, G, pi, scale):  # sum_t scale_t G_t score_t
    return d2l.tabular_score(b.obs, b.act, G * scale, pi).ravel()

grads = {'episodes': agg(b, G, pi_np, np.full(len(b), 1 / len(T))),
         'own length': agg(b, G, pi_np, np.repeat(1 / T, T) / len(T)),
         'total steps': agg(b, G, pi_np, np.full(len(b), 1 / len(b))),
         'a constant': agg(b, G, pi_np, np.full(len(b), 1 / 32))}
for k, u in grads.items():
    cos = u @ grads['episodes'] / (np.linalg.norm(u)
                                   * np.linalg.norm(grads['episodes']))
    print(f'{k:>12}: |grad| = {np.linalg.norm(u):.3f}, '
          f'cos to episodes = {cos:.3f}')
```

```
episode lengths [ 8 20  8 11], successes 3
    episodes: |grad| = 0.868, cos to episodes = 1.000
  own length: |grad| = 0.082, cos to episodes = 0.937
 total steps: |grad| = 0.074, cos to episodes = 1.000
  a constant: |grad| = 0.109, cos to episodes = 1.000
```

Three of the four gradients are parallel, at sizes an order of magnitude apart; the per-length version tilts away from the rest. The `policy_step` of Section 14.5 divides by the total number

of steps, the third choice, so the training loop below rescales its weights to recover the first. Language-model training faces the same choice with responses as episodes and tokens as steps: GRPO divides each response by its own length, DAPO divides by the total number of tokens (Q. Yu et al. 2025), and Dr. GRPO divides by a constant, the one choice that rescales every batch by the same factor (Z. Liu et al. 2025).

14.7.4 Five Estimators Compared

We now have five weightings for the score at step t. Which of them reduce the variance, and by how much? We answer first at a fixed policy, where the exact gradient is known, and then in training.

14.7.4.1 Estimators at a Frozen Policy

The five weightings are:

1. **Trajectory return** $R(\tau)$, the unbiased REINFORCE estimator displayed at the start of this section.
2. **Reward-to-go** $\hat{G}_t$ as in Equation 14.65, with the factor γ^t dropped.
3. **Centered** reward-to-go $\hat{G}_t - \bar{R}$, with $\bar{R}$ the mean over the steps of the batch.
4. **Normalized** reward-to-go Equation 14.71.
5. **Value baseline** $\hat{G}_t - V(s_t)$: at the fixed policy the exact V^{π_θ} from the linear solve, which we can afford here; in training the learned table $\hat{V}$ of Equation 14.70.

A sixth row, **causal**, weights the score at step t by $\gamma^t \hat{G}_t$, the unbiased estimator of the causality argument; it separates dropping past rewards from dropping γ^t.

Holding the probe fixed, we draw $2{,}000$ batches of four episodes and compute every estimator on each. `estimate` sums the weighted scores of a batch with `d2l.tabular_score` and divides by the number of episodes, as in Equation 14.65; `discounted` multiplies each reward-to-go by γ^t, counting t from the start of its episode; `weightings` maps each name to a function of the batch, its reward-to-go and a value table (`per_step` repeats each episode's return over its steps). `stats` computes the three printed columns. `|mean|/|grad J|` and `cos` give the scale and direction of the estimator's mean: the length of the mean of the draws relative to $\nabla_\theta J$, and the cosine between them. The column `noise` is the relative variance, the trace of the covariance of the draws divided by the squared length of their own mean. It is measured against each estimator's own mean because normalized weights are centered weights divided by σ, and a rescaled estimator should not be charged for its scale.

```
def estimate(b, w, pi):
    """One draw of the estimator: weighted scores, averaged over episodes."""
    return d2l.tabular_score(b.obs, b.act, w, pi).ravel() / len(b.ep_ends)

def discounted(b, G, gamma):
    """gamma^t G_t: reward-to-go with the factor gamma^t of the gradient kept."""
    t = np.concatenate([np.arange(ep.stop - ep.start) for ep in b.episodes()])
    return gamma ** t * G

weightings = {                  # V: exact V^pi here, the learned table later
    'return': lambda b, G, V: b.per_step(b.episode_returns(gamma)),
    'causal': lambda b, G, V: discounted(b, G, gamma),
```

```
    'reward-to-go': lambda b, G, V: G,
    'centered': lambda b, G, V: G - G.mean(),
    'normalized': lambda b, G, V: standardize(G),
    'value baseline': lambda b, G, V: G - V[b.obs]}

def stats(u, g):
    """The scale and direction of the mean of the draws u against g, and
    the variance of the draws relative to their own mean."""
    m = u.mean(axis=0)
    size_m, size_g = np.linalg.norm(m), np.linalg.norm(g)
    return (size_m / size_g, m @ g / (size_m * size_g),
            ((u - m) ** 2).sum(axis=1).mean() / (m @ m))
```

The lake has a single reward, at the goal, so there are no past rewards to drop. To see causality at work, the cell repeats the measurement with a charge of 0.1 for every move from a live cell. Every recorded step is such a move, so subtracting 0.1 from every reward of a batch gives the same episodes under the new reward; `costly` is the MDP with the same kernel and the charged reward, which supplies the exact gradient and V^{π}:

```
step_cost = 0.1                 # charged for every move from a live cell
live = ~np.isin(env.unwrapped.desc.ravel(), [b'H', b'G'])
costly = d2l.TabularMDP(mdp.P, mdp.r - step_cost * live[:, None], gamma)
cases = {'lake': (0.0, g_exact, V_pi),
         'step cost': (step_cost, exact_gradient(costly.r),
                       costly.policy_value(pi_np))}
rng = np.random.default_rng(4)
batches = [d2l.rollout(env, probe.act, 4, rng) for _ in range(2000)]
for case, (c, g, V) in cases.items():
    bs = [d2l.Batch(b.obs, b.act, b.rew - c, b.next_obs, b.term, b.ep_ends)
          for b in batches]
    for k, fn in weightings.items():
        u = np.stack([estimate(b, fn(b, b.reward_to_go(gamma), V), pi_np)
                      for b in bs])
        scale, cos, noise = stats(u, g)
        print(f'{case:>9} {k:>14}: |mean|/|grad J| = {scale:.2f}, '
              f'cos = {cos:.3f}, noise = {noise:5.1f}')
```

```
     lake         return: |mean|/|grad J| = 0.99, cos = 0.998, noise =  13.6
     lake         causal: |mean|/|grad J| = 0.99, cos = 0.998, noise =  13.6
     lake   reward-to-go: |mean|/|grad J| = 1.38, cos = 0.987, noise =  12.4
     lake       centered: |mean|/|grad J| = 1.11, cos = 0.986, noise =   7.6
     lake     normalized: |mean|/|grad J| = 3.28, cos = 0.982, noise =   7.5
     lake value baseline: |mean|/|grad J| = 1.38, cos = 0.990, noise =   5.7
step cost         return: |mean|/|grad J| = 0.98, cos = 0.996, noise =  19.5
step cost         causal: |mean|/|grad J| = 1.00, cos = 0.999, noise =   5.8
step cost   reward-to-go: |mean|/|grad J| = 1.29, cos = 0.989, noise =   5.4
step cost       centered: |mean|/|grad J| = 1.07, cos = 0.985, noise =   5.6
step cost     normalized: |mean|/|grad J| = 2.12, cos = 0.980, noise =   5.8
step cost value baseline: |mean|/|grad J| = 1.28, cos = 0.989, noise =   3.0
```

On the lake the return and causal rows agree draw for draw: with one reward at the end of a successful episode, $\gamma^t \hat{G}_t = \gamma^{T-1}$ is the return itself at every step. Reward-to-go dif-

fers from them only by the dropped γ^t, with a mean about 1.4 times longer and a slightly lower cosine. An unbiased estimator averaged over $2{,}000$ draws has a cosine near 0.997, so that column shows this tilt and little else. The noise column gives the ordering that holds when the draws are repeated with other seeds: the value baseline is quietest, centered and normalized come next and agree with each other, and return and reward-to-go are noisiest. Centering removes about 40 percent of the noise of reward-to-go, and the value baseline about half. Centering also shortens the mean to about 0.8 of that of reward-to-go, near the factor $(n-1)/n = 3/4$ of per-trajectory centering. Dividing by σ lengthens the mean about threefold and leaves the noise where centering put it, as a step size should.

With the charge per move, dropping past rewards cuts the noise of the return about threefold here. It need not. The dropped terms have mean zero but are correlated with the rest, and $\mathrm{Var}(X+Y) = \mathrm{Var}(X) + \mathrm{Var}(Y) + 2\,\mathrm{Cov}(X,Y)$ can be smaller than $\mathrm{Var}(X)$: at a charge of 0.01 per move, dropping them raises the noise by about a sixth (exercise 11). The ordering of the other rows also belongs to this policy; at the uniform policy centering does not help (exercise 7).

14.7.4.2 Step Size and Variance in Training

Training compounds many steps, and the step size enters. The arms below use the same seeds and differ only in the weight passed to `policy_step`. They update by plain SGD with step size η, because Adam rescales each coordinate and would partly undo the differences in scale under study. Every run updates $\widehat{V}$ by Equation 14.70, though only the value-baseline arm uses it. `entropy` averages the entropy $-\sum_a \pi(a \mid s) \log \pi(a \mid s)$, in nats, over the states of a batch, so each state counts as often as the agent visits it:

```
def entropy(theta, obs):
    """Entropy of pi(.|s) in nats, averaged over the states the batch visited."""
    logp = theta - np.logaddexp.reduce(theta, axis=1, keepdims=True)
    return float(-(np.exp(logp) * logp).sum(axis=1)[obs].mean())
```

`train` is REINFORCE by plain SGD with one of the `weightings` and a step size `eta`. Its other arguments are the environment and the settings fixed at the start of the section. `policy_step` averages over the N steps of a batch, while the frozen table averaged over its n episodes; multiplying every weight by N/n turns the first average into the second, so each arm runs the estimator the table measured. Without the factor the effective step would be $\eta\, n/N$, which grows as episodes shorten. Given the MDP as `exact`, `train` replaces the learned table by the exact V^{π_θ} of the policy about to be updated, solved afresh before every update as at the frozen policy; we call this arm the *exact baseline*. Each update records the batch's success rate, the length of the parameter step and the entropy along the batch.

`table` copies the preference table out of the policy, `d2l.plain_sgd` (the online appendix "Utility Functions and Classes") replaces the policy's Adam optimizer by plain SGD, and `d2l.run_seeds` (the online appendix "Utility Functions and Classes") calls `train` once per seed and stacks the results:

```
def table(ac):
    """The policy's preference table, copied out to numpy."""
    return ac.policy.weight.detach().numpy().copy()
```

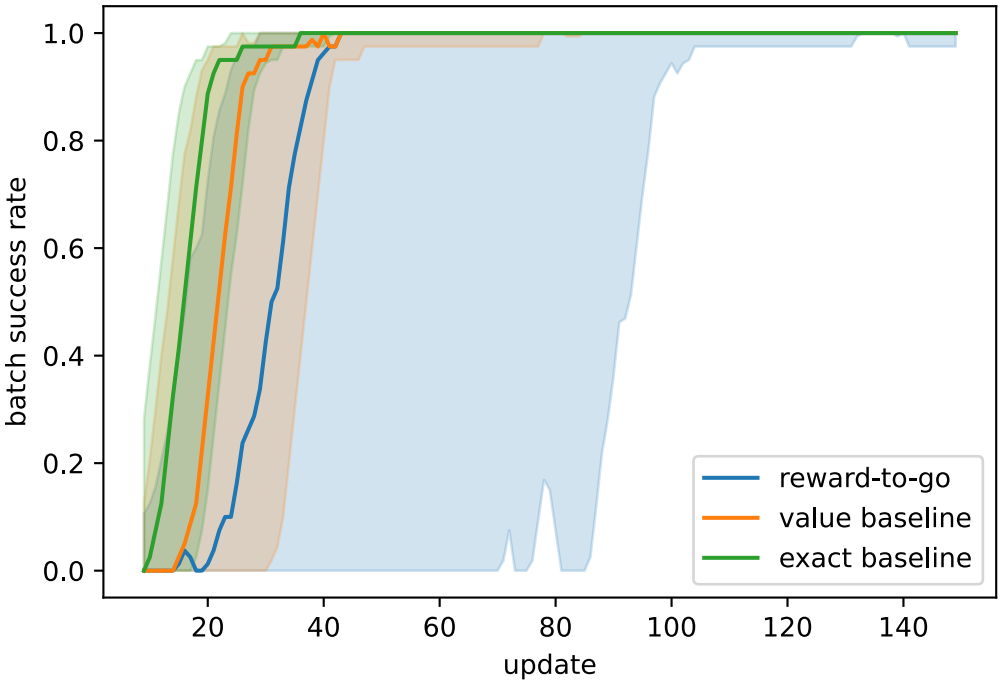
```
def train(seed, weighting, eta, env, gamma, num_updates, batch_episodes,
          eta_v, exact=None):
    """REINFORCE by plain SGD; the arms differ in one line, the weight. With
    an MDP as `exact`, V is the exact V^pi of the current policy."""
    S, A = env.observation_space.n, env.action_space.n
    rng, V = np.random.default_rng(seed), np.zeros(S, np.float32)
    ac = d2l.Policy.tabular(S, A, seed=seed)
    ac.opt_pi = d2l.plain_sgd(ac.policy, eta)
    env.reset(seed=seed)
    for _ in range(num_updates):
        batch = d2l.rollout(env, ac.act, batch_episodes, rng)
        G, before = batch.reward_to_go(gamma), table(ac)
        if exact is not None:              # V^pi of the policy that acted
            V = exact.policy_value(ac.probs(S, A))
        w = weighting(batch, G, V) * len(batch) / batch_episodes
        d2l.policy_step(ac, batch.obs, batch.act, w)  # sum over steps / n
        for s, g in zip(batch.obs, G):     # eq_value_baseline, every arm
            V[s] += eta_v * (g - V[s])
        yield (float(batch.episode_returns().mean()),
               float(np.linalg.norm(table(ac) - before)),
               entropy(before, batch.obs))
```

The weightings have different scales, so a shared η gives them steps of different lengths, and step length alone changes how fast a run learns. Matching the average step length does not remove this. A step's length includes its noise, and the steps of an arm with a baseline shrink once its policy succeeds, while an arm without one keeps pushing up the actions it already takes; arms matched on average therefore still take steps of different lengths while they learn. The clean test holds the expected update fixed instead. The exact V^{π_θ} is fixed before the batch is drawn, and so is the learned $\hat{V}$, which was fitted on earlier batches only. By the conditional identity, reward-to-go, the value baseline and the exact baseline therefore have the same expected update at every θ, and at one η they differ only in their noise. We run all six arms at $\eta = 2$ (`shared`) and these three again at $\eta = 16$ on `num_seeds_large` seeds (`large`). `setup` collects the settings every run shares, and each arm's runs are stacked into an array of shape (seeds, updates, 3):

```
variants = ['return', 'reward-to-go', 'centered', 'normalized',
            'value baseline']
arms = {v: dict(weighting=weightings[v]) for v in variants}
arms['exact baseline'] = dict(weighting=weightings['value baseline'],
                              exact=mdp)
setup = dict(env=env, gamma=gamma, num_updates=num_updates,
             batch_episodes=batch_episodes, eta_v=eta_v)
shared = {v: d2l.run_seeds(train, num_seeds, eta=2.0, **kw, **setup)
          for v, kw in arms.items()}
large = {v: d2l.run_seeds(train, num_seeds_large, eta=16.0, **arms[v],
                          **setup)
         for v in ('reward-to-go', 'value baseline', 'exact baseline')}
```

The success rate of the three arms with a common mean at $\eta = 16$, smoothed over a ten-update window, with each band spanning the interquartile range of the seeds around their median:

```
d2l.plot_curves({v: r[:, :, 0] for v, r in large.items()}, xlabel='update',
                ylabel='batch success rate', smooth=10, band='iqr')
```



`to90` finds the first update at which the success rate over the last ten updates reaches 90 percent; the tolerance absorbs the rounding of nine in ten. The cell prints, for both step sizes, the median over seeds with the fastest and slowest seed, the number of runs that never reach 90 percent within the 150 updates, and the mean step length over the updates before a run reaches 90 percent:

```
def to90(curve):
    """Last update of the first 10-update window whose mean success is 0.9."""
    hit = np.convolve(curve, np.ones(10) / 10, 'valid') >= 0.9 - 1e-9
    return hit.argmax() + 9 if hit.any() else len(curve)

for name, runs in (('eta = 2', shared), ('eta = 16', large)):
    print(name)
    for v, r in runs.items():
        reach = np.array([to90(c) for c in r[:, :, 0]])
        climb = np.mean([c[:t, 1].mean() for c, t in zip(r, reach)])
        print(f'{v:>14}: to 90%: median {np.median(reach):5.1f} '
              f'[{reach.min():3d}, {reach.max():3d}], never '
              f'{(reach == num_updates).sum():2d}, |step| {climb:.2f}')
```

```
eta = 2
        return: to 90%: median  63.0 [ 42, 108], never  0, |step| 0.43
  reward-to-go: to 90%: median  51.0 [ 29, 109], never  0, |step| 0.53
      centered: to 90%: median  58.5 [ 39, 119], never  0, |step| 0.32
```

```
    normalized: to 90%: median  32.0 [ 15,  85], never  0, |step| 0.93
value baseline: to 90%: median  52.0 [ 38,  97], never  0, |step| 0.37
exact baseline: to 90%: median  50.5 [ 33,  96], never  0, |step| 0.40
eta = 16
  reward-to-go: to 90%: median  35.0 [ 11, 150], never 15, |step| 1.46
value baseline: to 90%: median  26.0 [ 11, 150], never  5, |step| 1.30
exact baseline: to 90%: median  20.5 [ 10,  41], never  0, |step| 1.68
```

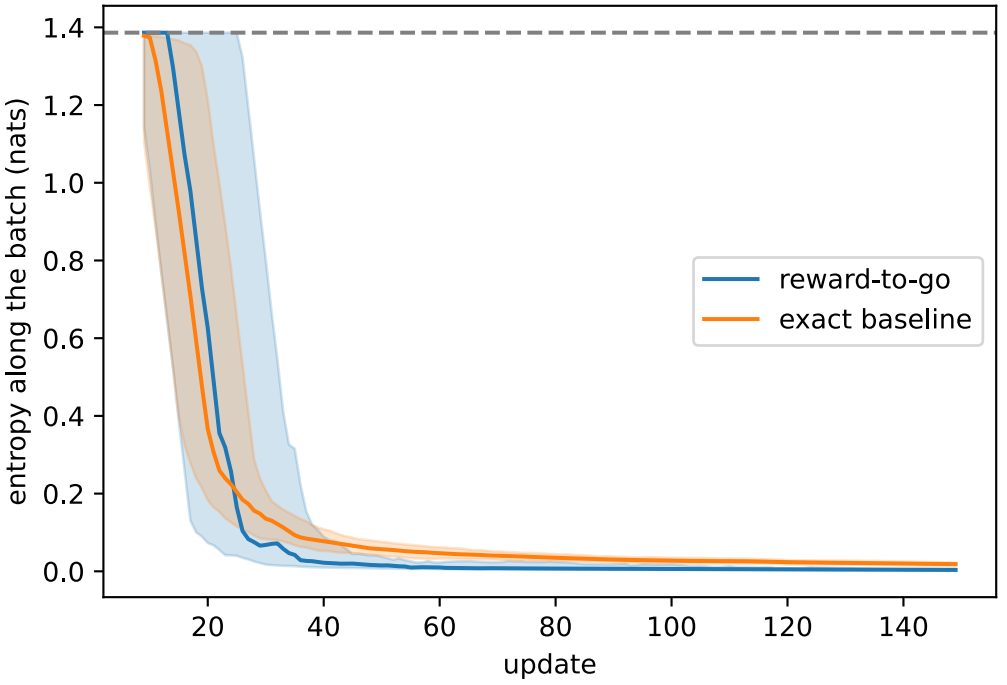
At the shared step size the ordering is a step-size effect. Normalization needs the fewest updates, about 30 against 50 to 70, but before reaching 90 percent its steps are about three times as long as those of centering and more than one and a half times those of reward-to-go. Centering is no faster than the plain return and slower than reward-to-go, the weight it centers, because it shrinks the step. The three arms with a common mean cannot be told apart at this step size: each needs about 50 updates (over 200 seeds outside the notebook, 51 for reward-to-go and the value baseline and 50 for the exact baseline).

At $\eta = 16$ they separate. All three learn faster than at $\eta = 2$, but the exact baseline gains most: over 200 seeds its median falls from 50 to 20 updates, and that of reward-to-go from 51 to about 30. The larger difference is in the failures. About one reward-to-go run in eight never reaches 90 percent, against none of the 200 exact-baseline runs. Nearly every such run has stalled: long steps along noisy directions made the policy nearly deterministic on a path that loops until the time limit or ends in a hole, every later episode returns zero, every weight is zero, and the policy never moves again. The learned table falls between the two (over 200 seeds, a median of 26 updates, and one run in twenty never reaches 90 percent), because at this step size the policy changes faster than $\hat{V}$ can follow. The step column shows that the gain is not a longer step: before reaching 90 percent the exact baseline moves at most about a sixth farther per update than reward-to-go. Lower variance does not make a small step better, since many small steps average the noise away; it makes a longer step usable (exercise 8).

At $\eta = 2$ the slowest seed of every arm needs two to six times the updates of the fastest, and at $\eta = 16$ the runs without the exact baseline spread wider still. Twenty seeds can therefore reorder medians that lie close together. Drawing twenty of the 200 seeds at random, the medians of reward-to-go and the exact baseline at $\eta = 2$ differ by more than six updates about one time in seven, while at $\eta = 16$ the exact baseline has the lower median in about 49 draws of 50, but a single reward-to-go seed that changes course can move the median of twenty by more than fifteen updates. The arms at $\eta = 16$ therefore run 100 seeds. We report medians and ranges, and check a claim on more seeds before relying on it (Henderson et al. 2018; R. Agarwal et al. 2021).

How deterministic does the policy become along the states the agent visits?

```
d2l.plot_curves({v: large[v][:, :, 2] for v in ('reward-to-go',
                                                'exact baseline')},
                xlabel='update', ylabel='entropy along the batch (nats)',
                reference=np.log(4), smooth=10, band='iqr')
```



At $\eta = 16$ the median of both arms falls from the uniform policy's $\ln 4 \approx 1.39$ nats to below 0.1 nats along the batch within the first 20 to 40 updates, that of reward-to-go usually sooner. A stalled run is as deterministic as a successful one, near zero nats on the path it repeats. An average over all sixteen states would stay above 0.4 nats, because the four holes and the goal are never acted in and keep $\ln 4$.

We can now say which weights may be used, how much variance each removes, and how to tell a better estimator from a longer step. Every estimator here indexed a table, sixteen rows of preferences and sixteen entries of $\widehat{V}$. A cart on a track is described by four real numbers and never visits the same state twice; Section 14.8 replaces both tables by networks.

14.7.5 Summary

The score of a policy's own action has zero mean, and so does its product with anything fixed before the action. This lets us drop earlier rewards and subtract a baseline without changing the expected gradient. A baseline is a control variate; the state value V^{π}, which turns the weight into a sampled advantage, removed about half the noise at the policy we measured. Centering each trajectory's return by the batch or group mean multiplies the expected gradient by $(n-1)/n$, which leave-one-out removes; centering step by step shrinks it by a different factor. Dividing by the spread changes the step size, and with binary rewards the group-normalized weight of GRPO pushes failures down; a group of one cannot learn. A shared step size favors whichever weighting takes the longest steps. Compared at one step size and with the same mean, the exact value baseline and reward-to-go learn equally fast at a small step, while at a large step the baseline learns faster and does not stall.

14.7.6 Exercises

1. **Admissible baselines.** The conditional zero-mean identity allows any quantity that is already determined when the agent stands at s_t to be subtracted from $\hat{G}_t$. For each candidate below, decide whether subtracting it in Equation 14.66 leaves the expected gradient unchanged, and justify the answer with that condition:

   1. the time index t;
   2. the sum $r_0 + \cdots + r_{t-1}$ of the rewards collected before step t;
   3. the number $T - t$ of steps that remain in the episode;
   4. a function of the next state s_{t+1};
   5. an action-value estimate $\hat{Q}(s_t, a_t)$.

   For the last candidate with $\hat{Q} = Q^{\pi_\theta}$, compute the expected contribution of step t after the subtraction. What does the result say about baselines that depend on the action?

2. **Baselines on stale data.** The zero-mean lemma averages the score under the same policy that is being differentiated. Suppose instead that the action at state s was drawn from an older policy $\pi_{\theta_{\text{old}}}$, as happens when a batch is reused for a second update, while the score is still $\nabla_\theta \log \pi_\theta(a \mid s)$.

   1. Use the softmax score $\partial \log \pi_\theta(a \mid s)/\partial \theta_{s,a'} = \mathbf{1}(a' = a) - \pi_\theta(a' \mid s)$ to compute, for a constant b,

   $$\mathbb{E}_{a \sim \pi_{\theta_{\text{old}}}(\cdot \mid s)}\big[b \, \nabla_\theta \log \pi_\theta(a \mid s)\big]. \tag{14.75}$$

   Which values of b introduce no bias, and how does the bias depend on the difference between the two policies?

   2. Show that multiplying each term by the ratio $\pi_\theta(a \mid s)/\pi_{\theta_{\text{old}}}(a \mid s)$ restores the zero mean for every b. Methods that update more than once on the same batch build on this ratio.

3. **Optimal constant baseline.** Fix a state s, write $\psi = \nabla_\theta \log \pi_\theta(a \mid s)$ for the score of the sampled action, and consider the single-step estimator $(\hat{G} - b)\,\psi$ with a constant b. Its mean does not depend on b, so we measure its noise by the trace of its covariance,

   $$\mathbb{E}\big[\|(\hat{G} - b)\,\psi\|^2\big] - \big\|\mathbb{E}[\hat{G}\,\psi]\big\|^2. \tag{14.76}$$

   1. Minimize the trace over b and show that the minimizer is

   $$b^* = \frac{\mathbb{E}\big[\|\psi\|^2 \, \hat{G}\big]}{\mathbb{E}\big[\|\psi\|^2\big]}. \tag{14.77}$$

   Show that any other b adds exactly $\mathbb{E}\big[\|\psi\|^2\big]\,(b - b^*)^2$ to the trace, and that b^* is the coefficient c^* of Equation 14.67 for $X = \hat{G}\,\psi$ and $Y = \psi$, with inner products in place of products.

2. In the one-step problem of Figure 14.14, the action is $a \sim \mathcal{N}(\mu, 1)$ at $\mu = 0$, so the score with respect to μ is a. Compute b^* for $R(a) = 0.4 + 2e^{-(a-2)^2/2}$ and confirm the value $b^* = 1.18$. The product of the standard normal density with $e^{-(a-2)^2/2}$ is proportional to another normal density. Why does b^* exceed $\mathbb{E}[R] = 0.92$?

4. code **Baseline step size and timing.** ⊠ The value-baseline arm of `train` updates its table $\hat{V}$ by Equation 14.70 after `policy_step`, so the baseline used on each batch was fitted on earlier batches only.
   1. Sweep η_V (`eta_v`) over $\{0.01, 0.1, 0.5, 1\}$ at $\eta = 16$, the step size of `large`, and report the median and the range over twenty seeds of the number of updates to 90% success. How sensitive is the arm to η_V?
   2. Move the value update above the line that computes the weights `w`, so that the baseline of each batch has already seen that batch, and repeat the sweep. At $\eta_V = 1$, what is the weight $\hat{G}_t - \hat{V}(s_t)$ of the last visit to each state in the batch?
   3. Explain both sweeps with the conditional zero-mean identity. Which of its conditions does the second ordering violate, and why does the damage grow with η_V?

5. code **Bias of batch centering.** ⊠ Section 14.7.3.2 obtains the factor $(n-1)/n$ by which per-trajectory batch centering shrinks the expected gradient from the leave-one-out identity $R_i - b_i = \frac{n}{n-1}(R_i - \bar{R})$. That argument applies only to per-trajectory weights.
   1. Derive the factor directly, without the leave-one-out baseline. Write the centered estimate as $\frac{1}{n}\sum_{i=1}^{n}(R(\tau_i) - \bar{R})\,\mathbf{S}_i$, where $\mathbf{S}_i$ is the summed score of trajectory i and $\bar{R}$ the mean return of the batch, expand $\mathbb{E}[\bar{R}\,\mathbf{S}_i]$ into its n terms, and use the independence of the trajectories. Which of the terms have a nonzero expectation?
   2. The centered arm of `train` instead subtracts the mean of all reward-to-go values in the batch, step by step. Extend the enumeration to this weighting. Each trajectory of the two-state problem has $\hat{G}_0 = r_0 + r_1$ and $\hat{G}_1 = r_1$, since the enumeration uses no discount, and row s of `S[i]` is the score of the step taken at state s. Compute the expected estimate for $n \in \{2, 4, 8\}$ and compare it with `g` through the ratio of the norms and the cosine.
   3. Explain the comparison. For each of the two steps, which reward-to-go values inside the batch mean have a nonzero expected product with the score of that step?

6. **The σ-normalized objective.** ⊠ GRPO divides the centered rewards of each group by their standard deviation σ, a choice that Z. Liu et al. (2025) analyze. Consider a prompt x with binary reward, $r = 1$ for a correct response and $r = 0$ otherwise, on which the current policy succeeds with probability p_x. Take the group so large that $\bar{R}$ and σ equal their population values p_x and $\sqrt{p_x(1-p_x)}$.
   1. Compute σ for $p_x = 0.5$ and for $p_x = 0.9$. By what factor does the division amplify the update of the second prompt relative to the first?
   2. Let y denote the sampled response. Show that

$$\mathbb{E}\big[(r - p_x)\,\nabla_\theta \log \pi_\theta(y \mid x)\big] = \nabla_\theta\, p_x. \tag{14.78}$$

Conclude that the σ-normalized update ascends $\sum_x h(p_x)$ for a function h with $h'(p) = 1/\sqrt{p(1-p)}$, and find h.

3. Compare h with the objective $\sum_x p_x$ of the unnormalized update. Which prompts gain weight relative to it? Is that the weighting a practitioner wants when most prompts of a training set are either nearly always or nearly never solved?

4. With a finite group of K responses, a prompt contributes no gradient when all K rewards agree. Compute the probability of this event for $p_x = 0.9$ and $K = 8$, and explain how it limits the amplification of prompts with small σ.

7. code **Variance ordering across policies.** ▯ The frozen-policy comparison of Section 14.7.4.1 probes one policy, `probe` after sixteen updates, at which $J(\theta) = 0.313$. Repeat it at the policies reached after $k \in \{0, 8, 32, 64\}$ probe updates: change the number of iterations of the loop that trains `probe`, and recompute `theta_probe`, `pi_np`, `g_exact` and `V_pi` before drawing the $2{,}000$ batches.

   1. Tabulate the noise of the five estimators against k. Does the ordering found at sixteen updates hold at every policy? At which policy does centering stop helping, and why?
   2. How do the two estimators without a baseline behave as $J(\theta)$ approaches its maximum $\gamma^5 \approx 0.77$, and why?
   3. At $k = 0$, count how many of the batches contain no success, and explain what such a batch contributes to each of the five estimators.

8. code **The step-size confound.** ▯ At the shared step size the normalized arm of `train` reached 90% success in the fewest updates, but it also took the longest steps. On a batch with $\sigma > 0$, the normalized weights Equation 14.71 produce the SGD update of the centered weights at the step size $\eta/(\sigma + 10^{-8})$.

   1. Log σ per batch in the normalized arm and plot the median over seeds of $1/(\sigma + 10^{-8})$ against the update index, skipping batches without a success. How does this multiplier of the step size change as the policy improves? What does that imply for an attempt to equalize the two arms by running the normalized arm at η divided by the run average of $1/(\sigma + 10^{-8})$?
   2. Repeat the comparison of reward-to-go, the value baseline and the exact baseline at $\eta \in \{4, 8, 32\}$, and plot for each arm, against η, the median number of updates to 90% and the fraction of runs that never get there. Which step size is best for each arm? Explain why the best step size grows as the noise of the estimator falls.
   3. Matching step lengths instead gives a noisier arm a smaller η. Choose for each of the five arms of `shared` a step size at which its mean step length over the first 40 updates is about 0.6, and log the mean step length separately before and after each run reaches 90%. Which arms take longer steps while they learn, and why does a comparison at matched average step lengths still mix variance with step size?

9. code **Optimal state baseline.** ▯▯ Exercise 3 derived the variance-optimal constant baseline at one state. Applied state by state, it gives the optimal baseline function of Greensmith et al. (2004), which minimizes the variance of a single step's term. This problem works it out for the tabular softmax policy and asks when it is also optimal for the episodic estimate `estimate`.

   1. Show from the softmax score that

$$\|\nabla_\theta \log \pi_\theta(a \mid s)\|^2 = 1 - 2\pi_\theta(a \mid s) + \sum_{a'} \pi_\theta(a' \mid s)^2. \tag{14.79}$$

Use $\mathbb{E}[\widehat{G}_t \mid s_t = s, a_t = a] = Q^{\pi_\theta}(s, a)$ to write the optimal baseline $b^*(s)$ in terms of $\pi_\theta(\cdot \mid s)$ and $Q^{\pi_\theta}(s, \cdot)$.

2. The estimate of one episode sums the weighted scores of all its steps. Show that in the tabular parameterization the scores of two steps at different states are orthogonal, so that the squared norm of the estimate splits into one term per step plus cross terms between visits to the same state. Under which condition on the episodes does $b^*(s)$ minimize the trace of the covariance of `estimate` exactly? Does FrozenLake meet it?

3. Compute $b^*(s)$ at `probe`, with Q^{π_θ} given by `mdp.backup(V_pi)`, and add the weighting `G - b_star[b.obs]` to `weightings` in Section 14.7.4.1 as `lambda b, G, V: G - b_star[b.obs]`. Compare its noise with that of the exact baseline, at `probe` and at the policy after 64 probe updates.

4. At which states does $b^*(s)$ differ most from $V^{\pi_\theta}(s)$? Explain with the weights $\|\nabla_\theta \log \pi_\theta(a \mid s)\|^2$ why a nearly deterministic state pulls $b^*(s)$ toward the action values of its rarely chosen actions, and relate the gap between the two baselines to the excess variance found in problem 3a.

10. [extended] **Length bias at small scale.** ⊠⊠ Dividing the loss of each episode by its own length changes the direction of the gradient (Section 14.7.3.4), and GRPO divides the loss of each response by its own length. Z. Liu et al. (2025) argue that this favors short correct responses and long incorrect ones. This project derives the effect in a toy model and looks for it on FrozenLake.

    1. Take a single prompt with three responses and a softmax over one logit per response: one correct response with reward 1 and length L_c, and two incorrect responses with reward 0 and lengths $L_s < L_l$. Weight the score of each response by $(r - p)/L$, where p is the probability of the correct response and L the length of the response. When the two incorrect responses are equally likely, compute the expected change of the logit difference $z_l - z_s$ after one ascent step of size η, and compare it with the change under a constant divisor.
    2. On FrozenLake, sample batches of eight episodes and weight every step of episode i by its centered return $A_i = R(\tau_i) - \frac{1}{8}\sum_{j=1}^{8} R(\tau_j)$, divided either by the episode's own length T_i or by the constant 10, where $R(\tau_i)$ is the discounted return `b.episode_returns(gamma)`. Since `policy_step` averages over all steps of the batch, pass the weight $A_i N/(8T_i)$, or $A_i N/80$, at every step of episode i, where N is `len(b)`, the number of steps in the batch. Train twenty seeds with each divisor for 200 updates of plain SGD at $\eta = 16$ (both divisors are about ten times larger than the per-episode divisor of `train`, so $\eta = 16$ here acts roughly like $\eta = 1.6$ there, close to the step size of `shared`; log the step lengths to check), and record per update the batch success rate and the mean length of the failed episodes.
    3. The two divisors learn at different speeds, so compare them at matched success rates: group the updates of all seeds into bins of success rate of width 0.2 and plot the mean length of failed episodes per bin for both divisors. Does FrozenLake

show the predicted bias? What does the same mechanism do to the incorrect responses of a language model during GRPO training?

11. code **Reward-to-go, taken apart.** ▣ The reward-to-go weight of Equation 14.65 makes two changes to the trajectory return: it drops the rewards collected before each action, which the conditional identity allows, and it drops the factor γ^t, which it does not.
    1. Let $\rho(s) = \mathbb{E}\big[\sum_t \mathbf{1}(s_t = s)\big]$ be the expected number of visits to s in one episode. As in problem 8a, the mean of Equation 14.65 is $\sum_s \rho(s) \sum_a \nabla_\theta \pi_\theta(a \mid s)\, Q^{\pi_\theta}(s,a)$, while the policy gradient weights the states by $\sum_t \gamma^t \Pr(s_t = s)$ instead. For which γ do the two agree?
    2. At `probe`, compute ρ by solving $\rho = \mathbf{e}_{s_0} + P_\pi^\top \rho$ over the eleven cells that are neither holes nor the goal, take Q^{π_θ} from `mdp.backup(V_pi)`, and evaluate the mean of Equation 14.65 exactly. Compare its cosine with `g_exact` and its length relative to `g_exact` with the reward-to-go row of the frozen-policy table.
    3. Dropping the earlier rewards removes a term Y of mean zero from an estimate $X+Y$. Show that $\mathrm{Var}(X) < \mathrm{Var}(X+Y)$ exactly when $\mathrm{Var}(Y) + 2\,\mathrm{Cov}(X,Y) > 0$. Repeat the comparison of the return and causal rows with a charge of 0.01 per move instead of 0.1, and at the policy reached after 64 probe updates. Does causality always reduce the noise? Explain each case by the sign of the covariance between the past rewards and the reward-to-go.

14.8 Function Approximation in Reinforcement Learning

For seven sections every quantity we learned lived in a table: sixteen rows for the cells of the lake, one row for each prompt of the verifier task. Now picture a cart that balances a pole hinged on top of it. Its state is four real numbers, the cart's position and velocity and the pole's angle and angular velocity, and in a thousand episodes the agent will never see the same four numbers twice. A table over such states would need a new row for every visit, and what it learned at one visit would say nothing about the next. We want a function that gives similar values to similar states, so that what the agent learns while the pole leans left at one speed carries over to the moment it leans left at a slightly different speed.

The problem is as old as the field. Bellman named the growth of tables with the number of state variables the *curse of dimensionality* (Bellman 1957b), and in 1959 Arthur Samuel's checkers program already scored board positions by a weighted sum of features instead of looking them up (Samuel 1959). Replacing a table by a parameterized function is called *function approximation*. With neural networks as the functions, it carried reinforcement learning to backgammon (Tesauro 1995), to Atari games (Mnih et al. 2015) and to Go (Silver et al. 2016).

For policy gradients the step is short, because nothing in the derivations of Section 14.6 and Section 14.7 used the table. REINFORCE needs only the score $\nabla_\theta \log \pi_\theta(a \mid s)$, and a network supplies it as readily as a table row. What changes is that one update now moves the estimate at every state. This sharing makes learning on continuous states possible, and it is also a new way for learning to go wrong. We make the step on two small control tasks, a cart with a pole and a pendulum driven by a continuous torque, measure how far one update spreads, and end with what the resulting agent still cannot do.

```
import gymnasium as gym
```

14.8.1 Networks in Place of Tables

The tables of the earlier sections become networks in two places: the policy, and the value estimate that served as a baseline. Before writing either network we look at a table in a form that a network extends.

14.8.1.1 Tables as Linear Networks on One-Hot States

`Policy.tabular` of Section 14.5 stores its preferences in a 16×4 embedding table θ, and the preferences at state s are row s of that table. Row s equals $\theta^\top \mathbf{e}_s$, where $\mathbf{e}_s$ is the one-hot vector of state s. A table is therefore a linear network whose input features are fixed one-hot vectors. The classic remedy for continuous states has the same form: it cuts the state space into boxes and lets $\mathbf{e}_s$ be the one-hot vector of the box that contains s. An early learning controller for the cart and pole below used 162 such boxes, a partition designed by hand (Barto et al. 1983). A network with one hidden layer keeps this form and replaces the fixed features by learned ones: the preferences become $\mathbf{W}_2 \tanh(\mathbf{W}_1 \mathbf{s} + \mathbf{b}_1) + \mathbf{b}_2$, where $\mathbf{s}$ is now the vector of real-valued observations itself.

The two forms differ in what one update does. Take a value estimate $\hat{V}_w$ with parameters w and one gradient step of size η_V on $\frac{1}{2}\big(y - \hat{V}_w(s_0)\big)^2$, which moves the estimate at a single state s_0 toward a target y. To first order in η_V, the estimate at any state s changes by

$$\Delta \hat{V}(s) = \eta_V \big(y - \hat{V}(s_0)\big)\, k(s, s_0), \qquad k(s, s_0) = \nabla_w \hat{V}(s)^\top \nabla_w \hat{V}(s_0). \tag{14.80}$$

Thus the change at s is the change requested at s_0, scaled by how much the gradients at the two states overlap. For a table, $\nabla_w \hat{V}(s) = \mathbf{e}_s$, so $k(s, s_0)$ is 1 when $s = s_0$ and 0 otherwise: a gradient step at one state leaves every other entry where it was. (Adam, the default optimizer of `Policy`, can still move an entry whose gradient is zero in the current batch, through its stored momentum, so the statement holds for gradients rather than for every optimizer step.) For a network, k is a smooth function of the two states, which is the *tangent kernel* of the network (Jacot et al. 2018). A step at s_0 then moves the estimates at similar states along with it. Exercise 7 derives Equation 14.80, and Section 14.8.4.1 measures the shape of k for the network we train.

14.8.1.2 The Policy and Value Networks

The `Policy` container of Section 14.5 holds a policy network and its optimizer. In Section 14.7 the learned baseline was a separate NumPy table: after each batch it moved $\hat{V}(s_t)$ a step η_V toward the reward-to-go $\hat{G}_t$ observed at every visited state. Here the baseline becomes a network too, and it moves into the container. `ActorCritic` subclasses `Policy` and adds a value module `value` with its own Adam optimizer `opt_v`. `V` returns the values as a framework array, the form a loss differentiates, and `value_np` takes and returns NumPy arrays, like the boundary methods of `Policy`. The name refers to the two parts: the policy is often called the *actor* and the value estimate the *critic*. Its `tabular` constructor builds both as zero-initialized tables, so that a table can stand in for either network, as in the measurement at the end of this section:

```
class ActorCritic(d2l.Policy):
    """A policy and a value function, each with its own optimizer."""
```

```
    def __init__(self, policy, value, lr=1e-2):
        super().__init__(policy, lr)
        self.value = value
        self.opt_v = torch.optim.Adam(value.parameters(), lr=lr)

    def V(self, obs):
        """V(s) for a batch of states, as a tensor."""
        return self.value(obs).squeeze(-1)

    def value_np(self, obs):
        """V(s) for a numpy batch of states, as numpy."""
        with torch.inference_mode():
            return self.V(torch.as_tensor(obs)).numpy()

    @classmethod
    def tabular(cls, num_states, num_actions, lr=0.1, seed=None):
        """Preference and value tables: two zero-initialized embeddings."""
        with d2l.seeded(seed):
            policy, value = (nn.Embedding(num_states, num_actions),
                             nn.Embedding(num_states, 1))
        nn.init.zeros_(policy.weight), nn.init.zeros_(value.weight)
        return cls(policy, value, lr)
```

The constructor `mlp` supplies networks instead of tables, each built by `tanh_mlp`: a policy network with `obs_dim` inputs and one preference per action, and a value network of the same shape with one output. Each has one hidden layer of 64 units, a common width for tasks this small, with tanh, the activation that a large study of on-policy methods found to work best for policy networks (Andrychowicz et al. 2021). The softmax of Section 14.6, $\pi_\theta(a \mid s) \propto \exp(\text{preference}_a)$, acts on the network outputs as it did on a table row. The score now includes the hidden layer, and automatic differentiation applies the extra chain rule:

```
def tanh_mlp(in_dim, hidden, out_dim):
    """A network with one tanh hidden layer."""
    return nn.Sequential(nn.Linear(in_dim, hidden), nn.Tanh(),
                         nn.Linear(hidden, out_dim))

@d2l.add_to_class(ActorCritic)
@classmethod
def mlp(cls, obs_dim, num_actions, hidden=64, lr=1e-2, seed=None):
    """The same container with the tables replaced by one-hidden-layer nets."""
    with d2l.seeded(seed):
        policy, value = (tanh_mlp(obs_dim, hidden, num_actions),
                         tanh_mlp(obs_dim, hidden, 1))
    return cls(policy, value, lr)
```

14.8.1.3 Value Regression

For a table, the update of the learned baseline moved one entry toward one observed target. For a network, the counterpart is regression: optimizer steps on the mean squared error between $\widehat{V}(s_t)$ and the targets of the whole batch. `fit_value` takes `num_steps` such steps toward a fixed target vector.

How many steps does the baseline need per batch? The targets are discounted returns of order 100 in size on both tasks below, while an Adam step moves each parameter by roughly its learning rate, 0.01. With one step per batch, the baseline lags behind its targets. On the pendulum task below, the fraction of the targets' variance that the baseline accounts for, $1 - \mathrm{Var}(\hat{G} - \hat{V})/\mathrm{Var}(\hat{G})$, then climbs only to between 0.4 and 0.8 by the end of a run on most seeds, and stays lower on some. The rest of the weight $\hat{G}_t - \hat{V}(s_t)$ is noise that a good baseline would remove. Forty steps per batch raise this fraction to about 0.99, and forty is what the training loop below uses.

```
def fit_value(ac, obs, target, num_steps=1):
    """Regress the value head on a fixed target: eq_value_baseline for nets."""
    obs, target = torch.as_tensor(obs), torch.as_tensor(target)
    for _ in range(num_steps):
        loss = ((ac.V(obs) - target) ** 2).mean()
        ac.opt_v.zero_grad()
        loss.backward()
        ac.opt_v.step()
    return loss.item()
```

14.8.1.4 Time Limits

Both tasks of this section stop episodes by a clock: CartPole after 500 steps, Pendulum after 200. Section 14.1 drew the consequence: the sum of rewards of an episode cut by a time limit is not its return, because the process would have gone on. Reward-to-go treats the cut as the end. Consider a CartPole episode that balances the pole until the limit. Near the cut its reward-to-go shrinks toward 1, while the value network sees a balanced pole like any other and predicts about what it predicts at upright states early in an episode. The weights $\hat{G}_t - \hat{V}(s_t)$ of the last steps are then strongly negative, and the update pushes down the very actions that kept the pole up. Take $\gamma = 0.99$, $T = 500$ and a value network that predicts $1/(1-\gamma) = 100$, the value of balancing forever. At $t = 490$ the reward-to-go is $\hat{G}_t = (1 - 0.99^{10})/0.01 \approx 9.6$, a weight near -90. Exercise 1 computes these weights along the whole episode.

The correction adds back the value of the future that the cut hid (Pardo et al. 2018). This treats the limit as a convenience of training, not as part of the task; if the limit were part of the task, the remaining time would belong in the state (Section 14.1). Let an episode have steps $t = 0, \dots, T-1$, let s_T be the state its last action reached, and let $\mathrm{term} \in \{0, 1\}$ record whether that state is terminal. The target of step t becomes

$$y_t = \hat{G}_t + (1 - \mathrm{term})\, \gamma^{T-t}\, \hat{V}(s_T).$$

For an episode that terminated nothing is added. For a truncated one, every step receives the discounted value of the state where recording stopped, as if the episode had continued from there. At $t = 490$ above the correction adds $0.99^{10} \cdot 100 \approx 90.4$, so $y_t \approx 100$ and the weight returns to about 0. The `Batch` of Section 14.6 already records what this needs: `next_obs` holds s_{t+1} for every step, and `term` records termination and never truncation, the rule of Section 14.1. `truncation_bootstrap` builds a vector that is zero except at the last step of each episode, where it holds $\gamma(1 - \mathrm{term})\hat{V}(s_T)$, and runs the backward scan $x_t + \gamma x_{t+1} +$

$\gamma^2 x_{t+2} + \cdots$ of Section 14.7 over it, which carries that value back to every earlier step with the factor γ^{T-t}:

```
@d2l.add_to_class(d2l.Batch)
def truncation_bootstrap(self, gamma, V):
    """gamma^(T-t) (1 - term_T) V(s_T) for every step t of each episode:
    the discounted value of the future that a time-limit cut did not see.
    V maps a batch of numpy states to numpy values."""
    tail = np.zeros_like(self.rew)
    last = self.ep_ends - 1
    tail[last] = gamma * (1 - self.term[last]) * V(self.next_obs[last])
    return self.backward_scan(tail, gamma)
```

14.8.2 REINFORCE on CartPole

With the policy network, the value network and the targets in place, we assemble the training loop and run it on the cart with the pole.

14.8.2.1 The Training Loop

`train_reinforce` is REINFORCE with a learned baseline for any policy object: `make_agent(seed)` builds the agent and `env_name` names the environment. Each update collects eight episodes with `rollout`, computes the targets y_t, takes one policy step with weights computed from $y_t - \hat{V}(s_t)$, and then fits $\hat{V}$ to the targets with forty regression steps. Apart from the networks, four things differ from the learned-baseline runs of Section 14.7. The optimizer is Adam, the one each `Policy` owns, instead of plain SGD. The weights pass through `d2l.standardize`, which subtracts their batch mean and divides by their batch standard deviation. The baseline is fitted by regression instead of entry by entry. Truncated episodes are bootstrapped.

Under Adam, dividing every weight by one positive number barely changes the step, because Adam divides each coordinate by a running estimate of its own scale (exercise 5 makes this precise). Normalization still does two things. Its centering acts as a second baseline, computed from the batch itself. Its division keeps the scale of the weights steady while the returns grow over a run, a drift that Adam's slowly decaying average follows only with a lag.

The update that results is not the gradient of the objective $J(\theta)$. With N steps in the batch and normalized weights $\tilde{w}_t$, the policy step follows

$$\hat{g} = \frac{1}{N} \sum_t \tilde{w}_t \, \nabla_\theta \log \pi_\theta(a_t \mid s_t), \qquad \tilde{w} = \text{standardize}\big(y - \hat{V}(s)\big). \tag{14.81}$$

Four things separate $\hat{g}$ from $\nabla_\theta J$. The factor γ^t is dropped, so the score at a late step counts as much as at an early one; this is the estimator Equation 14.65 of Section 14.7, which in general is not the gradient of any objective (Nota and Thomas 2020). The targets y_t of a truncated episode contain $\hat{V}(s_T)$, so the critic's error at the cut enters the weights as bias. The centering uses the batch's own mean, which depends weakly on the sampled actions. Dividing by N and by the batch's standard deviation rescales each batch by its own data, after which Adam takes a step of roughly fixed size. No stochastic-approximation theorem

applies to this combination directly, so the evidence that the loop works comes from runs, not from a convergence proof.

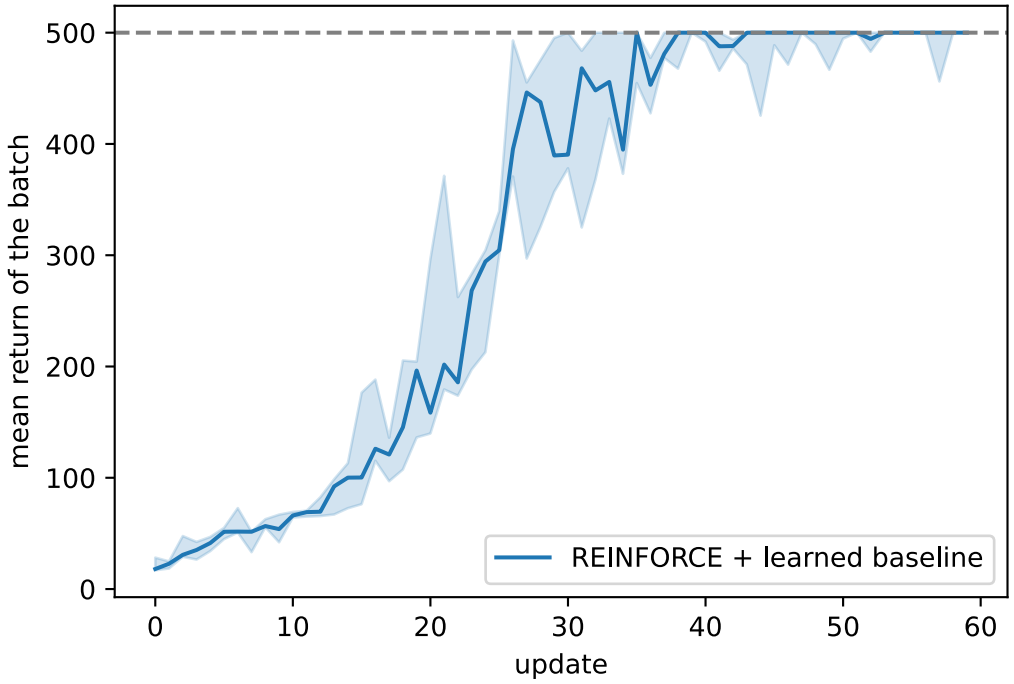
```
def train_reinforce(seed, make_agent, env_name, gamma=0.99, num_updates=60,
                    batch_episodes=8, value_steps=40):
    """REINFORCE with a learned baseline for the agent `make_agent` builds."""
    rng, env = np.random.default_rng(seed), gym.make(env_name)
    ac = make_agent(seed)
    env.reset(seed=seed)
    for _ in range(num_updates):
        batch = d2l.rollout(env, ac.act, batch_episodes, rng)
        G = (batch.reward_to_go(gamma)
             + batch.truncation_bootstrap(gamma, ac.value_np))   # time limits
        w = d2l.standardize(G - ac.value_np(batch.obs))
        L = d2l.policy_step(ac, batch.obs, batch.act, w)
        fit_value(ac, batch.obs, G, num_steps=value_steps)  # fit V-hat well
        yield float(batch.episode_returns().mean()), L
```

14.8.2.2 CartPole

CartPole is the classic pole-balancing problem (Barto et al. 1983). Its state is the four numbers of the opening, and its two actions push the cart left or right. Every step with the pole up earns reward 1. The episode terminates when the pole tips too far or the cart leaves the track, and it is truncated after 500 steps, so the best possible return is 500. Compared with the runs on the lake, three settings change: the constructor is `ActorCritic.mlp(4, 2)` instead of `Policy.tabular(16, 4)`, the environment is `'CartPole-v1'`, and the discount is 0.99, which raises the effective horizon $1/(1-\gamma)$ of Section 14.1 from 20 to 100 steps. `d2l.run_seeds` (the online appendix "Utility Functions and Classes") runs the training generator once per seed and stacks what it yields. We train three seeds for 60 updates:

```
def cartpole_agent(seed):
    return ActorCritic.mlp(4, 2, seed=seed)

runs = d2l.run_seeds(train_reinforce, 3, make_agent=cartpole_agent,
                     env_name='CartPole-v1')
d2l.plot_curves({'REINFORCE + learned baseline': runs[:, :, 0]},
                xlabel='update', ylabel='mean return of the batch',
                reference=500)
```



The untrained policy balances the pole for about 20 steps. Every seed's batch mean reaches the 500-step ceiling by about update 50 and mostly stays there, although single batches still dip below it. On this task the time-limit correction barely changed the average return in our runs, because only the successful episodes are truncated, but without it the policy left the ceiling more often late in the run.

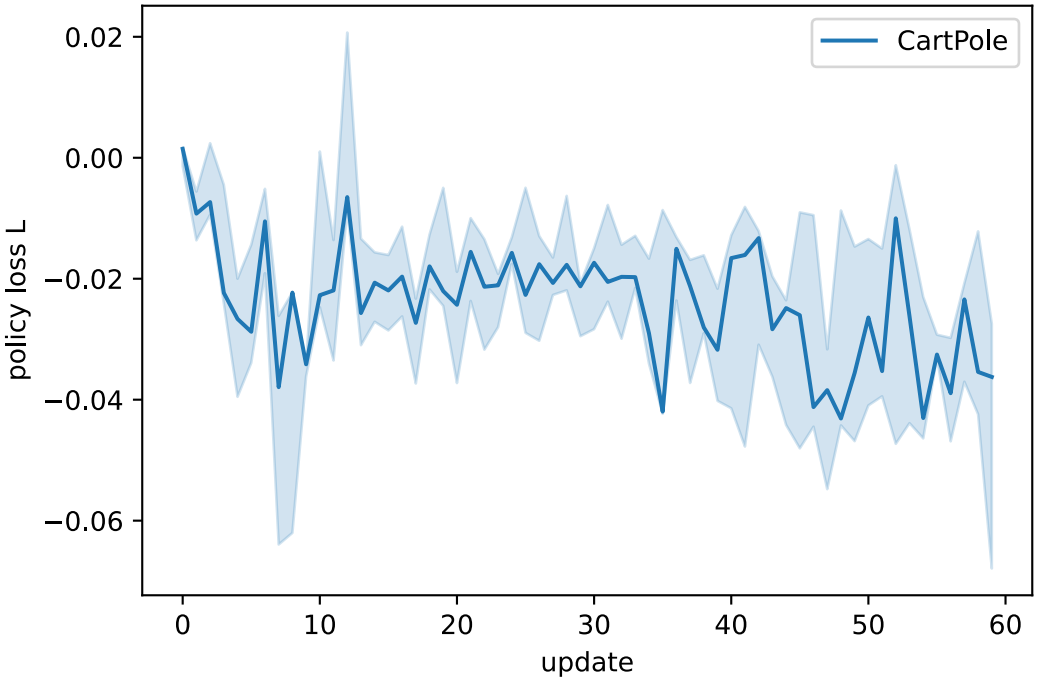
14.8.2.3 The Surrogate Loss

`train_reinforce` yields a second number at every update, the value `L` of the surrogate loss that `policy_step` minimizes (Section 14.6). Deep learning frameworks express updates through scalar losses, so instead of assembling $\hat{g}$ explicitly, `policy_step` minimizes

$$L(\theta) = -\frac{1}{N}\sum_{t} \tilde{w}_t \, \log \pi_\theta(a_t \mid s_t), \tag{14.82}$$

with the weights $\tilde{w}_t$ held fixed. They arrive as a NumPy array and carry no gradient graph. The gradient of L is $-\hat{g}$, so a descent step on L is a step along $\hat{g}$. The value of L, however, does not estimate how well the policy performs. L is built to have the right derivative at the current parameters, and nothing more, so a decrease in L need not mean an increase in return. The CartPole runs recorded both:

```
d2l.plot_curves({'CartPole': runs[:, :, 1]}, xlabel='update',
                ylabel='policy loss L')
```



While the return rises from 20 to 500, L stays small, below 0.1 in size, and negative on almost every update after the first. Because the normalized weights have mean zero and unit spread, L is minus the standard deviation of the log-probabilities in the batch times their correlation with the weights (exercise 4). It is negative when the actions with positive weight already have above-average probability, and its size says nothing about the return. To follow training, watch the return, and quantities that describe the update itself, such as how far the policy moved.

14.8.3 Continuous Actions

A softmax needs a finite list of actions. Many control problems have a real-valued action instead, a torque or a steering angle. The score-function derivation of Section 14.6 never used the finiteness of the action set; it needs only that $\log \pi_\theta(a \mid s)$ can be differentiated in θ. So a policy whose density over real actions has that property can be trained by the same loop.

14.8.3.1 The Gaussian Policy

Pendulum is a small task with a continuous action. A pendulum hangs from a motorized pivot, and the action is a torque $u \in [-2, 2]$. We write ϑ for the angle of the pendulum from upright, keeping θ for the policy parameters. The observation is $(\cos\vartheta, \sin\vartheta, \dot\vartheta)$, and each step gives the reward $-(\vartheta^2 + 0.1\,\dot\vartheta^2 + 0.001\,u^2)$ with ϑ wrapped to $[-\pi, \pi]$, so the cost is lowest when the pendulum stands still at the top. Every episode is truncated after 200 steps. An uncontrolled pendulum collects a return near -1200, and a controller that swings it up and balances it collects about -200 or better.

We use a Gaussian policy $\pi_\theta(a \mid s) = \mathcal{N}(a; \mu_\theta(s), \sigma^2)$, whose network produces the mean

action. `GaussianHead` is that network. A *trunk* computes hidden features, one tanh layer by default; the `trunk` argument replaces it by any module with `hidden` outputs. A linear layer maps the features to the mean $\mu_\theta(s)$; with `zero_mean=True` its weights and bias start at zero, which the Pendulum runs below use. The log standard deviation $\log \sigma$ is one learned vector by default, so σ is the same at every state. With `state_std=True` it becomes a second linear layer on the features, clamped to $[-5, 2]$, so that σ depends on the state; exercise 2 uses this option. Because $\log \sigma$ belongs to the policy module, the policy step updates it along with the mean:

```
class GaussianHead(nn.Module):
    """A Gaussian over actions: a trunk of hidden features, a linear mean,
    and a log standard deviation that is one learned vector or, with
    `state_std=True`, a second linear head clamped to [-5, 2]. The default
    trunk is one tanh layer; any module with `hidden` outputs will do.
    `zero_mean=True` sets the mean layer to zero, so that mu = 0 at first."""
    def __init__(self, obs_dim, act_dim, hidden, state_std=False, trunk=None,
                 zero_mean=False):
        super().__init__()
        self.trunk = trunk if trunk is not None else nn.Sequential(
            nn.Linear(obs_dim, hidden), nn.Tanh())
        self.mean = nn.Linear(hidden, act_dim)
        if zero_mean:
            nn.init.zeros_(self.mean.weight), nn.init.zeros_(self.mean.bias)
        self.state_std = state_std
        self.log_std = (nn.Linear(hidden, act_dim) if state_std
                        else nn.Parameter(torch.zeros(act_dim)))

    def forward(self, obs):
        h = self.trunk(obs)
        if self.state_std:
            return self.mean(h), self.log_std(h).clamp(-5, 2).exp()
        return self.mean(h), self.log_std.exp()
```

`GaussianPolicy` subclasses `ActorCritic`, builds a `GaussianHead` and a value network with `tanh_mlp`, like that of `mlp`, and overrides the three methods that depend on the distribution, `log_prob`, `act` and `act_greedy`. `rollout`, `policy_step`, `fit_value` and `train_reinforce` use only this interface and need no change:

```
class GaussianPolicy(ActorCritic):
    """The same interface over a Normal instead of a softmax; nothing that
    consumes the interface changes. `head_args` go to GaussianHead."""
    def __init__(self, obs_dim, act_dim, hidden=64, lr=1e-2, seed=None,
                 **head_args):
        with d2l.seeded(seed):
            policy = GaussianHead(obs_dim, act_dim, hidden, **head_args)
            value = tanh_mlp(obs_dim, hidden, 1)
        super().__init__(policy, value, lr)

    def log_prob(self, obs, act):
        """log pi(a|s): the Normal density, summed over action dimensions."""
        mean, std = self.policy(obs)
```

```
        return torch.distributions.Normal(mean, std).log_prob(act).sum(-1)

    def act(self, obs, rng):
        """Sample mu(s) + sigma(s) z, z ~ N(0, I) from the numpy rng."""
        with torch.inference_mode():
            mean, std = self.policy(torch.as_tensor(obs))
        return mean.numpy() + std.numpy() * rng.standard_normal(
            mean.shape, dtype=np.float32)

    def act_greedy(self, obs, rng=None):
        """The mean action mu(s)."""
        with torch.inference_mode():
            return self.policy(torch.as_tensor(obs))[0].numpy()
```

The environment clips each sampled torque to $[-2, 2]$, so `act` can return the unbounded Gaussian draw; `log_prob` scores the unclipped sample, and exercise 2 shows why the estimate stays unbiased. The update uses `log_prob` only through its gradient, the score. With $z = (a - \mu)/\sigma$, the score of a Gaussian is

$$\frac{\partial \log \pi}{\partial \mu} = \frac{z}{\sigma}, \qquad \frac{\partial \log \pi}{\partial \log \sigma} = z^2 - 1.$$

With a positive weight, a sample above the mean pulls μ up, a sample far from the mean in either direction pulls σ up, and a sample close to the mean pulls σ down. We check `log_prob` against the formula for the Gaussian density, and its gradients against these two formulas, at $\sigma = 0.5$ and five random states and actions. The mean layer's bias enters μ with derivative 1, so its gradient is the score in μ summed over the five samples:

```
sigma = 0.5
gp = GaussianPolicy(3, 1, seed=1)
gp.policy.log_std.data.fill_(np.log(sigma))
rng = np.random.default_rng(1)
obs = rng.standard_normal((5, 3)).astype(np.float32)
act = rng.standard_normal((5, 1)).astype(np.float32)
z = (act - np.stack([gp.act_greedy(o) for o in obs])) / sigma
hand = -0.5 * z ** 2 - np.log(sigma) - 0.5 * np.log(2 * np.pi)
print('density:', np.allclose(gp.log_prob_np(obs, act), hand.sum(-1), atol=1e-5))
gp.log_prob(torch.as_tensor(obs), torch.as_tensor(act)).sum().backward()
g_mu, g_ls = gp.policy.mean.bias.grad.numpy(), gp.policy.log_std.grad.numpy()
print('score in mu:', np.allclose(g_mu, (z / sigma).sum(0), atol=1e-4),
      '| score in log sigma:', np.allclose(g_ls, (z ** 2 - 1).sum(0), atol=1e-4))
```

```
density: True
score in mu: True | score in log sigma: True
```

All three checks print `True`: `log_prob` is the Gaussian density, and its gradients are the score that the update uses.

14.8.3.2 Pendulum

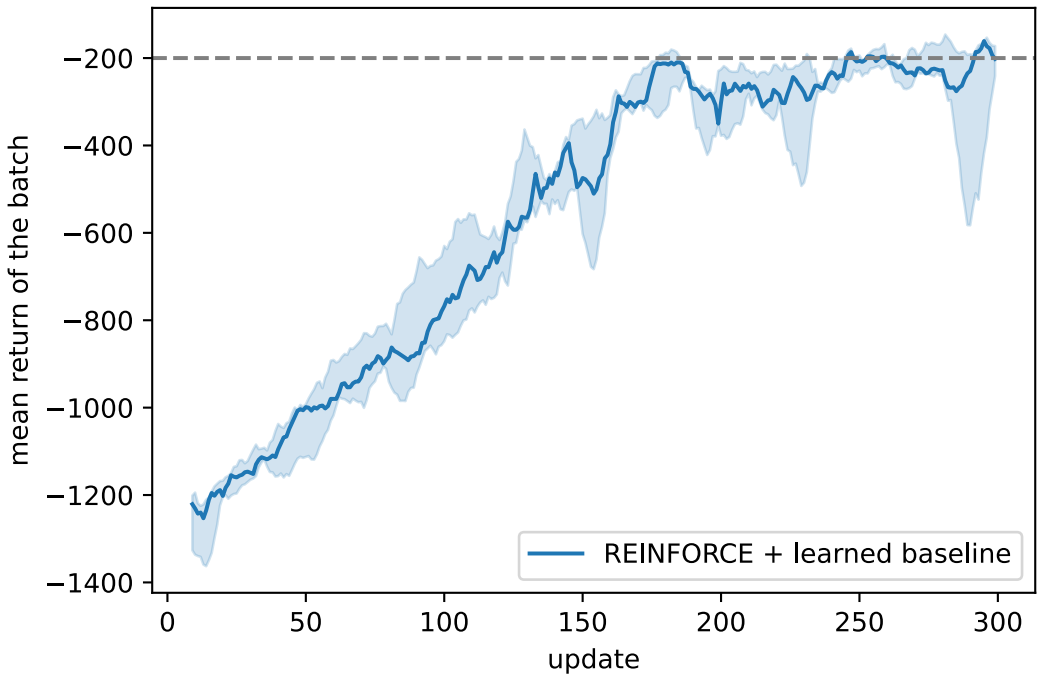
The curves report the undiscounted 200-step return, while the update discounts at $\gamma = 0.95$, an effective horizon of 20 steps. The discount is our choice here, not part of the task. A

torque matters for the next few dozen steps far more than for the cost a hundred steps later, and the shorter horizon gives targets with lower variance but some bias with respect to the reported objective; $\gamma = 0.9$ trained about as well in our runs.

`pendulum_agent` sets the weights and the bias of the mean layer to zero (`zero_mean=True`), so that every seed starts at $\mu = 0$ with $\sigma = 1$, whatever the random draw of the other layers. A mean outside $[-2, 2]$ would stall learning. The environment would clip nearly every sample to the same bound, the reward would no longer depend on which sample was drawn, and the score would be multiplied by weights that carry no information about the action. The initialization of our layers keeps the initial mean well inside the bounds, but initializations with larger weights put it outside on some seeds, and those seeds stall. A last policy layer with very small or zero weights is a standard recommendation for on-policy methods (Andrychowicz et al. 2021). The policy must first learn to swing the pendulum up before it can balance it, so we train for 300 updates of eight episodes:

```
def pendulum_agent(seed):   # every seed starts at mu = 0
    return GaussianPolicy(3, 1, seed=seed, zero_mean=True)

runs_p = d2l.run_seeds(train_reinforce, 3, make_agent=pendulum_agent,
                       env_name='Pendulum-v1', gamma=0.95, num_updates=300)
d2l.plot_curves({'REINFORCE + learned baseline': runs_p[:, :, 0]},
                xlabel='update', ylabel='mean return of the batch',
                reference=-200, smooth=10)
```



Returns start near -1200. Every run learns to swing the pendulum up and balance it. Its ten-update average crosses -200 after roughly 150 to 300 updates, that is, after a few hundred thousand environment steps, and its best stays between about -150 and -200. Later updates mostly stay near that level, although a run can fall back by several hundred for a few

updates. Over the run σ falls to about half its initial value, the direction that exercise 2 predicts near a good action. Here the time-limit correction matters: every Pendulum episode is truncated, and without the correction only about half of the seeds in our runs reached -200 within the same budget.

A few hundred thousand steps is a great deal of experience for a task this small. Each batch of 1,600 steps drives one update and is then discarded, and the weights come from whole returns rather than from a learned estimate of each action's value. Soft actor-critic (Section 15.5) reaches -200 on the same task within about 5,000 to 11,000 steps. How a method can learn that much more from each sample is the question that Chapter 15 takes up.

14.8.3.3 Score-Function and Pathwise Gradients

A continuous action also allows a second way to differentiate an expected reward. Consider a single state, a Gaussian policy $a \sim \mathcal{N}(\mu, \sigma^2)$, and a known differentiable reward $Q(a)$. The gradient of $\mathbb{E}[Q(a)]$ with respect to μ has two equivalent forms:

$$\nabla_\mu \mathbb{E}[Q(a)] = \mathbb{E}\left[Q(a)\,\frac{a-\mu}{\sigma^2}\right] = \mathbb{E}[Q'(\mu+\sigma z)], \qquad z \sim \mathcal{N}(0,1). \tag{14.83}$$

The first equality is the score-function identity used throughout this chapter. The second substitutes $a = \mu + \sigma z$, which moves μ from the distribution into Q, and differentiates through the sample. This is the *pathwise* or reparameterization gradient (Kingma and Welling 2014), and Figure 14.15 shows where each estimator's gradient flows. With a known Q both are unbiased. We compare their noise on $Q(a) = -(a-1)^2$ at $\mu = 0$ and $\sigma = 0.5$, where the exact gradient is 2:

```
def Q(a):                    # a stand-in critic, differentiable in the action
    return -(a - 1.0) ** 2

sigma, N = 0.5, 100000
a = sigma * np.random.default_rng(2).standard_normal(N, dtype=np.float32)
g_score = Q(a) * a / sigma ** 2              # score-function samples, mu = 0
at = torch.as_tensor(a).requires_grad_(True)  # pathwise: grad through a
Q(at).sum().backward()
g_path = at.grad.numpy()                     # per-sample dQ/da (da/dmu = 1)
print(f'score:    mean {g_score.mean():.2f}, variance {g_score.var():.1f}')
print(f'pathwise: mean {g_path.mean():.2f}, variance {g_path.var():.2f}')
print(f'score variance if Q gains a constant +10: '
      f'{((Q(a) + 10) * a / sigma ** 2).var():.0f}; pathwise is unchanged')
```

```
score:    mean 1.98, variance 21.5
pathwise: mean 2.00, variance 1.00
score variance if Q gains a constant +10: 281; pathwise is unchanged
```

Both sample means are close to 2. The exact single-sample variances are 21.75 for the score function and 1.0 for the pathwise estimator, a ratio of about twenty, and adding the constant 10 to Q raises the score-function variance to 281.75, because Q multiplies the score. The pathwise estimator depends only on Q', so a constant does not change it. Exercise 6 derives these variances; the ratio depends on Q, on σ and on the offset. The price of the lower variance is access to Q as a differentiable function of the action. In practice Q is a learned

critic Q_w, and then the error of its action gradient $\partial Q_w/\partial a$ becomes bias in the policy gradient.

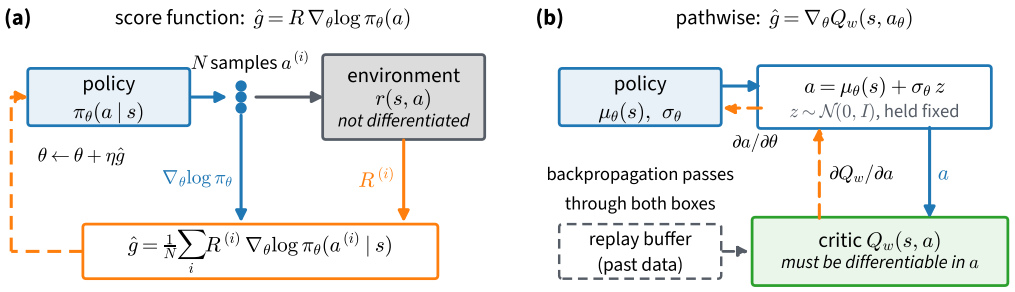


Figure 14.15: Two estimators of the same gradient. (a) The score-function estimator uses the environment only through the returned reward, so the environment need not be differentiable. (b) The pathwise estimator writes the sample as $a = \mu_\theta(s) + \sigma_\theta z$, here with a single learned σ_θ, and differentiates through $\partial Q_w/\partial a$ and $\partial a/\partial\theta$. The critic Q_w must be differentiable in the action. It can be trained by regression on replayed data, and its error in $\partial Q_w/\partial a$ becomes bias in the gradient.

The pathwise form is also how continuous control avoids a maximization over actions. A value-based method acts by computing $\max_a Q(s,a)$, a lookup among four actions on the lake but an optimization problem for $a \in \mathbb{R}^d$ at every step. The deterministic policy gradient instead trains an actor network to output a high-value action, following $\partial Q_w/\partial a$ through a learned critic (Silver et al. 2014).

14.8.4 Generalization across States

The opening asked for a function that gives similar values to similar states. Equation 14.80 says that one gradient step spreads across states according to the kernel k. We now measure that spread for a freshly built value network of the shape that trained CartPole.

14.8.4.1 A Probe of State Coupling

The probe raises $\widehat{V}$ at one of 256 random states in $[-1,1]^4$ by one gradient step toward a target one unit above its current estimate, and records the change at all 256 states. It then does the same for a table of sixteen entries. Each probe replaces the Adam optimizer `opt_v` by plain SGD (`d2l.plain_sgd`, the online appendix "Utility Functions and Classes"), so that the one step follows Equation 14.80; Adam would rescale every parameter by its own running scale. `fit_value` minimizes the mean squared error, whose gradient is twice that of the half squared error, so the table's step of 0.25 moves its entry by 0.5. The cell sorts the other 255 probes by their distance from the nudged state:

```
net_probe = ActorCritic.mlp(4, 2, seed=0)
tab_probe = ActorCritic.tabular(16, 4, seed=0)
for ac, lr in ((net_probe, 0.02), (tab_probe, 0.25)):  # one plain step
    ac.opt_v = d2l.plain_sgd(ac.value, lr)
S = np.random.default_rng(0).uniform(-1, 1, (256, 4)).astype(np.float32)
```

```
before = net_probe.value_np(S)
fit_value(net_probe, S[:1], before[:1] + 1.0)      # raise ONE state
dV = net_probe.value_np(S) - before
order = 1 + np.argsort(np.linalg.norm(S[1:] - S[0], axis=1))  # nearest first
before_t = tab_probe.value_np(np.arange(16))
fit_value(tab_probe, np.arange(1), before_t[:1] + 1.0)
dV_t = tab_probe.value_np(np.arange(16)) - before_t
print(f'network: nudged state {dV[0]:+.2f}; 10 nearest probes '
      f'{dV[order[:10]].mean():+.2f}, 10 farthest {dV[order[-10:]].mean():+.2f}; '
      f'{(dV[1:] < 0).sum()} of 255 moved down')
print(f'table:   nudged entry {dV_t[0]:+.2f}; '
      f'largest move among the other fifteen: {np.abs(dV_t[1:]).max():.6f}')
```

```
network: nudged state +0.49; 10 nearest probes +0.43, 10 farthest -0.06; 15 of 255 moved down
table:   nudged entry +0.50; largest move among the other fifteen: 0.000000
```

The nudged state rises, and the ten probes nearest to it rise almost as much, by more than 80 percent of its change. The change falls off with distance, and at the farthest probes it is near zero or slightly negative: some probes move the other way. The table moves the nudged entry by $+0.50$ and no other entry at all. How far down the farthest probes move, and how many probes move down, depends on the random initialization of the network; the shape does not. Figure 14.16 draws the same contrast on a one-dimensional state.

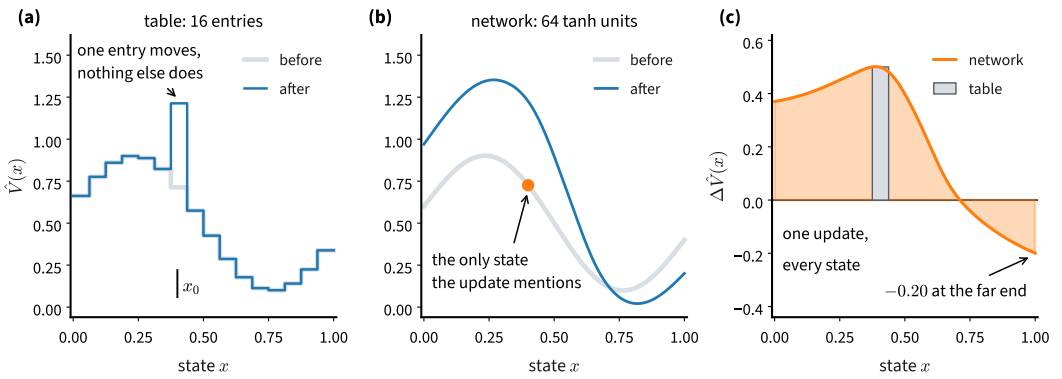


Figure 14.16: What one value update touches. A value estimate over a one-dimensional state is fitted as a sixteen-entry table and as a network with 64 tanh units, and each takes one gradient step toward a target one unit above its estimate at the marked state x_0, with the step size η_V of each chosen so that x_0 moves by $+0.50$. (a) The table moves one entry and nothing else. (b) The network moves its whole curve, because every state's estimate shares parameters with x_0's. (c) The two changes side by side. To first order the network's change is $\eta_V \left(y - \hat{V}(x_0)\right) k(x, x_0)$ for the target y, where k is the inner product of the parameter gradients at x and x_0: a bump around x_0 that turns negative beyond $x \approx 0.7$ and reaches -0.20 at the far end.

The positive bump is generalization: experience at one state informs the estimates at states like it, which is why a network learns CartPole without a hand-made partition of its state space. The negative far field is interference: fitting one region of the state space changes the estimates in regions the batch does not contain. A CartPole policy that balances well collects batches of nearly upright states, and fitting them moves the estimates at large pole

angles that the batch never visits (exercise 7 follows this through). Such changes can make a training curve dip even when every update fits its own batch well.

14.8.4.2 Limits of the Monte Carlo Agent

The value network of this section is fitted to observed returns, except at a time-limit cut, where the target contains its own estimate $\widehat{V}(s_T)$. Away from the cut, fitting it is ordinary regression. Methods that *bootstrap* at every step, putting the estimate at the next state into every target, lose this property. Their target moves with every update, the update is no longer the gradient of a fixed objective, and with nonlinear approximators or data from another policy the iteration can diverge (Tsitsiklis and Van Roy 1997).

We can now train a policy network on continuous states and continuous actions with the REINFORCE estimator, a fitted baseline and the time-limit correction. The agent still has three limits. It waits for every episode to end, because its targets are whole returns; Section 15.1 bootstraps to learn sooner. It uses each batch for one step and discards it; Section 15.2 reuses a batch for a few controlled steps, and Section 15.4 replays old transitions. It does not limit how far one update moves the policy, although a large move can spoil both the current return and the data collected next. Chapter 15 takes up all three, and with them the gap between a few hundred thousand steps here and ten thousand for soft actor-critic.

14.8.5 A Map of the Methods

This chapter and the next present many methods. Two questions locate each of them: what does it learn, and which data may drive its update? Figure 14.17 arranges the methods along these two questions.

Table 14.2 lists the algorithms of both chapters with what each estimates, which data it can use, and where it is introduced.

Table 14.2: The algorithm map of these two chapters. "Named in" marks methods this book describes but does not implement.

algorithm	what it estimates	which data may drive the update	where
value iteration, policy iteration	V^* exactly, by sweeping the Bellman operator; V^π by policy evaluation	none: it consumes the kernel P and reward r	Section 14.2
behavior cloning	$\pi(a \mid s)$ by cross-entropy on demonstrations	a fixed expert dataset	Section 14.5
DAgger	the same fit	the learner's own states, relabeled by an expert on call	Section 14.5
Q-learning (tabular)	Q^* from sampled backups	any behavior's transitions (off-policy)	Section 14.3

algorithm	what it estimates	which data may drive the update	where
SARSA	the behavior's own Q, exploration included	its own transitions (on-policy)	Section 14.3
certainty-equivalent planning	Q^* of the model counted from samples	any behavior's transitions, counted	Section 14.3
UCB, Thompson sampling	arm means plus their uncertainty	its own pulls (the bandit)	Section 14.4
REINFORCE	$\nabla_\theta J$ by the score function	fresh trajectories from the current policy only	Section 14.6
REINFORCE with baselines, RLOO	the same gradient at lower variance	fresh trajectories from the current policy	Section 14.7
GRPO	group-normalized advantages, no value network	fresh groups per prompt, each reused for a few clipped steps	weights in Section 14.7; machinery in Section 15.8
REINFORCE with a learned baseline, on networks	π_θ and $\hat{V}$ as networks	fresh trajectories from the current policy	Section 14.8
actor-critic (A2C)	π_θ plus a bootstrapped critic	fresh, near-current trajectories	Section 15.1
PPO	π_θ under a clipped probability ratio	one batch, reused for a few steps	Section 15.2
KL-regularized policy optimization (the objective behind RLHF)	a policy tilted from a reference by reward	rollouts scored by a learned reward	Section 15.3; RLHF at scale is left to the Language Models part, which takes it up in more depth
DQN, Double DQN	Q as a network	a replay buffer of stale experience	Section 15.4
Rainbow	DQN plus its measured components	a replay buffer	named in Section 15.9
DDPG, TD3	a critic, and a deterministic actor trained to maximize it	a replay buffer (off-policy)	named in Section 15.5
SAC	twin soft critics, and a stochastic squashed actor maximizing reward plus entropy	a replay buffer (off-policy)	Section 15.5

algorithm	what it estimates	which data may drive the update	where
model-predictive control in a learned model; search distilled into the policy	a model of the dynamics and reward fitted by regression, and plans searched in it	the agent's own stored experience, refit into the model	Section 15.6
Dyna, MBPO	the same model, used to imagine extra transitions for an off-policy learner	a replay buffer plus model rollouts	named in Section 15.6
offline Q-learning with pessimism, CQL	Q penalized where the data is thin	a fixed logged dataset, no interaction at all	Section 15.7
Decision Transformer	a return-conditioned sequence model	a fixed logged dataset	named in Section 15.7
MuZero, Dreamer	a learned model of the environment, to plan or imagine in	its own interaction, replayed through the model	named in Section 15.9
DPO	the regularized optimum directly from preferences	a fixed preference dataset	Section 15.8; preference optimization at scale is left to the Language Models part, which takes it up in more depth

14.8.6 Summary

A table over continuous states would need a row for every visit, so the policy and the value estimate become networks; a table is the special case of a linear network on one-hot states. The policy-gradient derivation needs only the score, so REINFORCE with a learned baseline carries over to networks with three practical additions: the baseline is fitted by enough regression steps to track its targets, the weights are normalized, and episodes cut by a time limit are bootstrapped with the value of their last state. The update this loop runs is the per-step surrogate rather than an unbiased gradient of J, and its loss value says nothing about performance. A Gaussian policy extends the loop to a continuous torque, and with it every Pendulum run swings up and balances, after a few hundred thousand steps. One update moves the estimate at every state, raising similar states and lowering distant ones. The agent waits for whole episodes, uses each batch once and does not limit its steps, which is where Chapter 15 begins.

14.8.7 Exercises

1. **Discounting at the time limit.** `train_reinforce` yields the undiscounted mean return of the episodes in each batch, while its update weights each step by the reward-to-go at $\gamma = 0.99$. Every CartPole step earns reward 1, and the environment truncates any episode that reaches 500 steps.

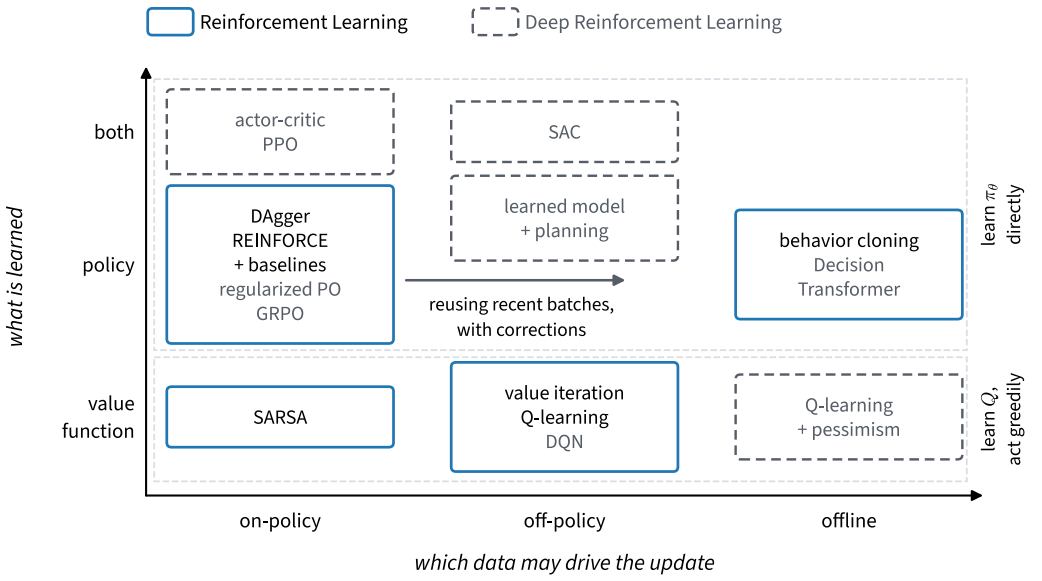


Figure 14.17: The methods of this chapter and of Chapter 15, arranged by what they learn and which data may drive their updates. Solid boxes hold this chapter's methods; dashed boxes and gray names belong to Chapter 15. Behavior cloning learns from a fixed set of demonstrations, while DAgger learns from the learner's own states relabeled by an expert, which places it in the on-policy column. Value iteration sits in the off-policy column because its update uses the known kernel rather than data from any one policy. The arrow from the on-policy policy box marks the reuse of recent batches, which needs a correction because the policy has changed since they were collected. The learned model in the off-policy column is fitted to the agent's own stored experience, and the agent plans in it (Section 15.6).

1. Show that the discounted return of an episode of length T is $(1-\gamma^T)/(1-\gamma)$, and evaluate it for $T \in \{20, 100, 200, 500\}$. Conclude that the undiscounted return and the discounted return rank episodes identically. By what factor does each grow when an episode lengthens from 200 to 500 steps?
2. Without the time-limit correction, compute the reward-to-go $\hat{G}_t$ at $t \in \{0, 400, 450, 490\}$ of an episode that reaches the limit. The value network sees only the state, and the states late in a successful episode resemble those early in it. Which sign do the weights of the last steps of a successful episode tend to have, and what does the policy update do to the actions taken at those steps?
3. `train_reinforce` adds `batch.truncation_bootstrap(gamma, ac.value_np)` to these targets. Suppose that $\hat{V}$ predicts $1/(1-\gamma)$ at every state with the pole up. Show that the target y_t of every step of a truncated episode is then $1/(1-\gamma)$, whatever its distance from the limit. Why does the target of an episode that ends with the pole falling still shrink toward its end, and why is that correct?
4. Every Pendulum episode is truncated, while on CartPole only the successful episodes are. Use this to explain why removing the correction slows Pendulum down much more than CartPole.

2. code **Gaussian score.** `GaussianHead` parameterizes a one-dimensional action by the mean μ and the log standard deviation $\ell = \log\sigma$, which is `log_std`. Write $z = (a-\mu)/\sigma$ for a sampled action a.
   1. Show that $\partial \log\pi/\partial\mu = z/\sigma$ and $\partial \log\pi/\partial\ell = z^2 - 1$, and that both have mean zero under the policy.
   2. Let Q be a quadratic function of the action with second derivative Q''. Show that $\mathbb{E}[Q(a)\,(z^2-1)] = \sigma^2 Q''$, using $\mathbb{E}[z^4] = 3$. In which direction does the expected policy-gradient step move σ near a maximum of Q? What does this imply for the exploration of a policy whose mean approaches the best action?
   3. Pendulum clips each sampled torque to $[-2, 2]$, while `log_prob` scores the unclipped sample. Explain why the score-function estimate remains unbiased for the return of the clipped environment.
   4. Build the agent of `pendulum_agent` with `GaussianPolicy(3, 1, state_std=True)`, so that σ depends on the state, and train three seeds with `train_reinforce` as in the Pendulum run. Every 25 updates, record σ at the hanging state $(\cos\vartheta, \sin\vartheta, \dot{\vartheta}) = (-1, 0, 0)$ and at the upright state $(1, 0, 0)$. Where does σ shrink first, and how does this agree with part b? Compare the return curve with `runs_p`.

   Adapted from Richard S. Sutton and Andrew G. Barto, Reinforcement Learning: An Introduction, Exercise 13.4.

3. code **Discontinuous rewards.** The second equality of Equation 14.83 differentiates through Q. Take $\mu = 0$ and $\sigma = 0.5$, as in the computation of `g_score` and `g_path`, and the threshold reward $Q(a) = 1$ for $a > 1$ and $Q(a) = 0$ otherwise.
   1. Compute the exact gradient $\nabla_\mu \mathbb{E}[Q(a)]$. What does the pathwise estimator return for every sample? Which step of the derivation of the second equality fails?
   2. Replace `Q` in that computation by the smooth step $Q_k(a) = 1/(1+e^{-k(a-1)})$ for $k \in \{1, 10, 100, 1000\}$, and record the mean and the variance of `g_score` and of `g_path`. The samples are single precision, so for $k \geq 100$ the factor $e^{-k(a-1)}$ overflows at the smallest samples and the derivative of the formula as written is NaN there; implement Q_k with a numerically stable sigmoid. Plot both variances against k. Which estimator has the lower variance at large k? Explain how the

variance of `g_path` depends on k through the height and the width of Q'_k, and how the variance of `g_score` depends on k through Q_k.

3. Pendulum clips each sampled torque to $[-2, 2]$. What does a pathwise estimate through the clipped torque return at samples outside this interval? Why does the score-function estimate of problem 2c not lose these samples?

4. **Surrogate loss value.** `policy_step` evaluates the surrogate Equation 14.82 on the batch that the current policy has just collected. The weights from `d2l.standardize` have mean 0 and standard deviation 1 over the N steps of the batch.

   1. Show that $L(\theta) = -s\,c$, where s is the standard deviation of $\log \pi_\theta(a_t \mid s_t)$ over the batch and c is the correlation coefficient between the weights and these log-probabilities. Conclude that $|L(\theta)| \le s$.
   2. For `GaussianPolicy` with one action dimension, show that $s \approx 1/\sqrt{2}$ for every mean network and every σ. What does this imply for the policy loss of the Pendulum runs?
   3. How does s behave on CartPole as the policy becomes nearly deterministic? What does this imply for the value of L late in training?

5. code **Normalization under Adam.** ☒ Under plain SGD, dividing the weights by their standard deviation changes the effective learning rate (exercise 8 of Section 14.7). `train_reinforce` updates with Adam instead. Its step for each coordinate is $-\eta\,\hat{m}_t/(\sqrt{\hat{v}_t} + \epsilon_0)$, where $\hat{m}_t$ and $\hat{v}_t$ are bias-corrected moving averages of the gradient g_t and of g_t^2 with decay rates $\beta_1 = 0.9$ and $\beta_2 = 0.999$.

   1. Show that for $\epsilon_0 = 0$, multiplying every gradient by the same constant $c > 0$ leaves every Adam step unchanged.
   2. Show that

$$\hat{v}_t = \frac{1-\beta_2}{1-\beta_2^t} \sum_{i=1}^{t} \beta_2^{t-i} g_i^2.$$

      By what factor do the weights of the oldest and the newest squared gradient differ at $t = 300$, the length of the Pendulum runs? Suppose that the gradient of a coordinate keeps a fixed sign and drops tenfold in scale halfway through a run of 300 updates. By what factor is the step at the end of the run smaller than the step at the start?

   3. Keep the centering of `d2l.standardize` in `train_reinforce` but drop its division, so that the weights are `A - A.mean()` with `A = G - ac.value_np(batch.obs)` and the two variants differ only in scale. Log the standard deviation of the weights of every batch, and train three seeds of both variants on CartPole and on Pendulum. How does the scale of the weights evolve on each task? On which task does part b predict that dropping the division changes the return curves more?

6. **Score versus pathwise variance.** ☒ The samples `g_score` and `g_path` estimate the two forms of Equation 14.83 for $Q(a) = -(a-1)^2$ at $\mu = 0$. Use $\mathbb{E}[z^4] = 3$ and $\mathbb{E}[z^6] = 15$ for $z \sim \mathcal{N}(0, 1)$.

   1. Show that the single-sample variance of the score-function estimate is $15\sigma^2 + 14 + 1/\sigma^2$ and that of the pathwise estimate is $4\sigma^2$. Evaluate both at $\sigma = 0.5$, and compare them with the printed variances.

2. Adding a constant c to Q is the same as subtracting the baseline $b = -c$. With the score a/σ^2 in the formula for b^* of exercise 3 of Section 14.7, show that $b^* = -(1 + 3\sigma^2)$ and that the smallest score-function variance is $6\sigma^2 + 8$. Check the printed variance for $c = 10$ against your formula.
3. Let $\sigma \to 0$. How do the two variances behave, with and without the optimal baseline? Explain the limit 8 of the score-function variance through the Taylor expansion of Q around μ. Which of the two estimators remains usable for a deterministic actor?

7. code **Tangent kernel.** ▣ The measurement of `dV` raises $\hat{V}$ at one probe state with one plain SGD step of `fit_value`. Write w for the parameters of the value network and

$$k(s, s') = \nabla_w \hat{V}(s)^\top \nabla_w \hat{V}(s') \tag{14.84}$$

for its tangent kernel, as in Equation 14.80.

1. Show that one gradient step of size η on $\frac{1}{2}\big(y - \hat{V}(s_0)\big)^2$ changes the estimate at s by $\eta\,\big(y - \hat{V}(s_0)\big)\,k(s_0, s) + O(\eta^2)$. `fit_value` minimizes $\big(y - \hat{V}(s_0)\big)^2$ instead; how does the prediction change?
2. For a freshly constructed `net_probe`, compute the gradients $\nabla_w \hat{V}$ at the 256 probe states `S` and the kernel values $k(s_0, s)$ for s_0 = `S[0]`. Take one plain gradient step with $\eta = 10^{-3}$ toward $y = \hat{V}(s_0) + 1$, without Adam, and compare the measured change at every probe with the prediction.
3. Plot $k(s_0, s)/k(s_0, s_0)$ against the distance $\|s - s_0\|$. How many probes have a negative kernel value, and how does that count compare with the number of probes that the cell reported moving down?
4. A CartPole policy that balances well collects batches of mostly near-vertical states. Use the kernel to explain how fitting such a batch changes the value estimate and the policy at large pole angles that the batch never contains. Why can that change make the policy worse at those angles?

8. [extended] **Parameter-space noise.** ▣▣ Policy gradients inject noise into the actions. Evolution strategies (Salimans et al. 2017) inject it into the parameters instead: they perturb the parameters $\theta \in \mathbb{R}^d$ by $\nu\mathbf{z}$ with $\mathbf{z} \sim \mathcal{N}(\mathbf{0}, I)$, run one episode with the perturbed deterministic policy, and ascend the smoothed objective

$$J_\nu(\theta) = \mathbb{E}_{\mathbf{z}}\big[J(\theta + \nu\mathbf{z})\big]. \tag{14.85}$$

1. Apply the score-function identity of Equation 14.83 to the distribution $\mathcal{N}(\theta, \nu^2 I)$ of the parameters, and show that

$$\nabla J_\nu(\theta) = \frac{1}{\nu}\,\mathbb{E}\big[J(\theta + \nu\mathbf{z})\,\mathbf{z}\big]. \tag{14.86}$$

Show that the antithetic estimate

$$\frac{1}{2\nu}\big(J(\theta + \nu\mathbf{z}) - J(\theta - \nu\mathbf{z})\big)\,\mathbf{z} \tag{14.87}$$

has the same mean. Which properties of the policy and of the environment does the estimate not require?

2. For a linear objective $J(\theta) = \mathbf{c}^\top \theta$, show that the antithetic estimate equals $(\mathbf{c}^\top \mathbf{z})\,\mathbf{z}$ and that its expected squared norm is $(d+2)\,\|\mathbf{c}\|^2$. How does the noise of this estimate scale with d and with the episode length? How do the same two quantities enter the noise of REINFORCE?

3. Implement the estimate on CartPole for the deterministic linear policy that pushes right when $\theta^\top s > 0$, so that $d = 4$. Use eight antithetic pairs per update and divide the step by the standard deviation of the sixteen returns, as in the random search of Mania et al. (2018) . Start from $\nu = 0.1$ and a step size of 0.02. Count the sixteen perturbed episodes of every update toward the budget, and measure progress by the mean return of the unperturbed policy over five further episodes. Plot this return against the number of episodes consumed for three seeds, together with `train_reinforce` for a policy network replaced by a single linear layer from the four state coordinates to the two action preferences, keeping the value network of `ActorCritic.mlp`, and with `runs`. Both of these consume eight episodes per update.

4. Repeat with a deterministic one-hidden-layer tanh network of width 64 whose single output selects the action by its sign, so that $d = 385$. How many episodes does each method need until its mean return first exceeds 400, with the small and with the large policy? How do these numbers change with d for each method? Which properties of CartPole favor the search in parameter space, and on which kinds of task would you expect the comparison to reverse?

Chapter 15

Deep Reinforcement Learning

The last agent of Chapter 14 swung a pendulum up after a few hundred thousand steps. A simulator makes that cheap. A robot arm would wear out first, and a language model trained on its own answers cannot discard each batch of answers after one update. The agent waited for every episode to end, used each batch once and never limited how far an update moved its policy. How much can one sample teach, and what goes wrong when we make it teach more?

Four ideas answer, each with an error of its own. An agent can trust a learned estimate of the rest of an episode instead of waiting for it (Samuel 1959; Sutton 1988); the estimate is biased while it is young. It can reuse an older policy's data, reweighted or replayed (Lin 1992); the data goes stale and the targets move. It can bound each update by the change in the policy's behavior (Kakade and Langford 2002; Schulman et al. 2017); the bound limits each step, not where the steps lead. It can learn a model of the environment and plan in it (Sutton 1990), inheriting the model's errors and computing at every step. Together, a learned target, a network shared across states and another policy's data can make values diverge, which a table never does. Controllers, game players and post-trained language models are built from these ideas.

Section 15.1 asks whether a policy must wait for the episode to end. Section 15.2 asks whether one batch can drive several updates, and how far. Section 15.3 asks how far a policy may stray from the data behind a learned reward. Section 15.4 asks whether a network can learn values from any transition, as a table can. Section 15.5 asks how the pendulum can be learned in a few thousand steps. Section 15.6 asks whether an agent that learns the physics from a few hundred steps can plan its way up. Section 15.7 asks what can be learned when no new action may be tried. Section 15.8 asks what survives when the environment appends tokens and a checker gives the reward.

CartPole carries the three methods with discrete actions, so that their sample counts share one axis, and Pendulum carries soft actor-critic and the learned model. Exact answers come from the verifier task and two gridworlds: one built so that a learned reward can be exploited, and the slippery lake. The code, in PyTorch and JAX, runs in CPU minutes. Figure 14.17 and Table 14.2 place every method of both chapters, and Section 15.9 collects those this chapter names without teaching. Several agents, hidden state, and meta- and hierarchical learning are left out; the Language Models part treats post-training at scale, including learning from human preferences, in more depth.

Resources and Further Reading

The resources of Chapter 14, among them CleanRL's single-file implementations and the catalog of PPO's implementation details, serve this chapter as well. The following accompany particular sections; Section 15.9 turns to the methods and topics the chapter does not teach.

- Chapters 11 to 13 of Sutton and Barto (Sutton and Barto 2018) treat the λ-return and eligibility traces of Section 15.1, the actor-critic methods built on them, and the deadly triad of Section 15.4 with what can be proved once one of its three ingredients is dropped. Their chapter 8 develops Dyna and planning with a learned model, the tabular ancestors of Section 15.6.
- The controlled study of Engstrom et al. (2020) accompanies Section 15.2: at matched code-level choices PPO and TRPO perform nearly alike, so much of PPO's practical edge comes from details its objective does not mention.
- The RL Baselines3 Zoo (Raffin et al. 2021) ships tuned hyperparameters per environment for the PPO, DQN and SAC of Section 15.2, Section 15.4 and Section 15.5, the starting point when one of them meets a task other than CartPole or Pendulum.
- rliable (R. Agarwal et al. 2021) provides interval estimates and performance profiles, the tools to compare results over a few seeds more carefully than the counts and medians this chapter quotes.
- The survey of Levine et al. (2020) continues Section 15.7 from the tabular diagnosis to the deep offline methods.
- The RLHF book (Lambert 2026) develops the post-training methods whose update Section 15.3 and Section 15.8 assemble, at the scale of real language models.

15.1 Actor-Critic Methods and Multi-Step Returns

The agent of Section 14.8 learns to balance the pole, but it learns late. Each of its updates waits for eight whole episodes, up to 4,000 steps of balancing, and only then asks of every action whether the rest of its episode went better than expected. The Q-learning agent of Section 14.3 needed no such patience: it changed one table entry after every transition, using its own estimate of what came next in place of the rest of the episode. Can a policy-gradient agent learn the same way?

It can, and the piece it needs is already in hand. The learned baseline $\hat{V}$ of Section 14.8 is trained to predict the reward-to-go, the discounted sum of the rewards that follow a state. If it predicts the rest of the episode, the agent need not wait for it: one reward plus the prediction at the next state can stand in for the sampled tail. The policy then becomes the *actor* and the value estimate its *critic*. Barto et al. (1983) built the first such pair for this very cart-pole problem, and the "adaptive critic element" of their paper gave the critic its name.

The substitution trades the randomness of the future for the error of the critic. This section works out that trade: what a single transition can tell the actor, how much of the sampled tail to keep, and how the trade comes out on CartPole, where the critic's error turns out to matter more than the variance it removes. Most methods in the rest of this chapter carry a critic of some kind, and the estimator built here, a geometric mix of one-step and many-step targets, is the one PPO uses.

```
import gymnasium as gym
```

```
torch.set_num_threads(1)
```

We keep the task and the agent of Section 14.8. CartPole's state is four numbers, the cart's position and velocity and the pole's angle and angular velocity; its two actions push the cart left or right; every step with the pole up pays 1; and an episode is cut off after 500 steps, so the largest possible return is 500. The discount is $\gamma = 0.99$. The agent is `ActorCritic.mlp`, a policy network and a value network with one hidden layer of 64 units each, each with its own Adam optimizer. The main training runs below take 150 updates of eight episodes each, up to 600,000 environment steps. The new hyperparameter `critic_steps` sets how many regression passes the critic takes per batch, and `grad_clip` caps the norm of the actor's gradient; both are explained with the training loop.

```
gamma, num_updates, batch_episodes = 0.99, 150, 8
num_seeds, critic_steps, grad_clip = 3, 20, 0.5
def cartpole_agent(seed):
    return d2l.ActorCritic.mlp(4, 2, seed=seed)
```

15.1.1 The Temporal-Difference Advantage

15.1.1.1 Bootstrapping the Reward-to-Go

To see what the critic can replace, write out one step of the reward-to-go:

$$\hat{G}_t = r_t + \gamma\, \hat{G}_{t+1}. \tag{15.1}$$

The learned baseline is regressed toward the reward-to-go, so $\hat{V}(s_{t+1})$ predicts the very quantity that $\hat{G}_{t+1}$ samples. Instead of waiting for the sample, we substitute the prediction, $\hat{G}_t \approx r_t + \gamma \hat{V}(s_{t+1})$. The weight on the score at step t was $\hat{G}_t - \hat{V}(s_t)$; after the substitution it becomes

$$\delta_t = r_t + \gamma\, \hat{V}(s_{t+1}) - \hat{V}(s_t), \tag{15.2}$$

with $\hat{V}(s_{t+1}) = 0$ when s_{t+1} is terminal. This *temporal-difference (TD) error* compares two predictions of the same return: one made at s_t, and one made a step later with a real reward in hand.

The TD error of Section 14.3 had the same form with one difference. Q-learning bootstraps on the best continuation, $\max_{a'} \hat{Q}(s', a')$, because it estimates the optimal values; here the bootstrap is the current policy's own continuation value, as in the SARSA target of that section. The terminal mask follows `terminated`, not `truncated`: a time limit stops the observation without ending the task, so the value of the state at a cut is not zero (Section 14.1). `td_target` computes the target $r_t + \gamma \hat{V}(s_{t+1})$ for every transition of a `Batch`, the flat record of transitions and episode boundaries that `d2l.rollout` fills (Section 14.6), from any function that maps states to values. It returns a NumPy array, so the critic's gradient does not flow through the target.

```
@d2l.add_to_class(d2l.Batch)
def td_target(self, bootstrap, gamma):
    """r_t + gamma (1 - terminated) V(s'), by a numpy bootstrap."""
    return self.rew + gamma * (1 - self.term) * bootstrap(self.next_obs)
```

As in the reward-to-go estimator Equation 14.65 of Section 14.7, the actor's update omits the factor γ^t that would discount late steps of an episode relative to early ones.

15.1.1.2 One Transition by Hand

Three numbers show what the new weight says. Suppose the critic predicts 100 at every state, the value $1/(1-\gamma)$ of balancing forever at $\gamma = 0.99$. On a step on which the pole stays up, $\delta_t = 1 + 0.99 \cdot 100 - 100 = 0$: the step went as predicted, and the action taken is neither encouraged nor discouraged. On the step on which the pole falls, the episode terminates, the bootstrap is masked, and $\delta_t = 1 - 100 = -99$: the action that ended the episode is pushed down hard.

A freshly initialized critic predicts values near 0 everywhere. Then every step gives $\delta_t \approx 1$, whether its action kept the pole up or dropped it. Every action taken is pushed up by the same amount, and the actor learns little until the critic can tell the two apart.

15.1.1.3 An Exact Critic Gives the Advantage

With a perfect critic, the TD error estimates the same quantity as the Monte Carlo weight. Suppose for a moment that $\hat{V} = V^\pi$. The action value satisfies $Q^\pi(s,a) = r(s,a) + \gamma \sum_{s'} P(s' \mid s,a)\, V^\pi(s')$, the Bellman equation of Section 14.2 written with V^π. Averaging Equation 15.2 over the next state therefore gives

$$\mathbb{E}\big[\delta_t \mid s_t, a_t\big] = r(s_t,a_t) + \gamma \sum_{s'} P(s' \mid s_t,a_t)\, V^\pi(s') - V^\pi(s_t) = Q^\pi(s_t,a_t) - V^\pi(s_t).$$

The right-hand side is the advantage $A^\pi(s_t,a_t)$, how much better the action taken is than the policy's average at that state. The learned baseline of Section 14.7 estimated it with $\hat{G}_t - \hat{V}(s_t)$. One transition, with one reward and two critic evaluations, gives an unbiased sample of the advantage, where the Monte Carlo weight needed the entire rest of the episode.

During training $\hat{V}$ is not equal to V^π, and then δ_t is a biased estimate of the advantage. The zero-mean identity of Section 14.7 says that anything fixed before the action may be subtracted from the weight without moving the gradient's mean. It covers $\hat{V}(s_t)$, which depends on the current state only. It does not cover $\hat{V}(s_{t+1})$, which depends on the action through the next state, so the critic's error at the states the actions lead to enters the update as bias (exercise 1). This bias comes on top of the tilt that dropping γ^t already gives Equation 14.65.

The critic's own update is sampled policy evaluation. Policy evaluation in Section 14.2 applied the Bellman expectation backup, whose fixed point is V^π. Regressing $\hat{V}(s_t)$ toward $r_t + \gamma \hat{V}(s_{t+1})$ replaces that expectation by one sampled transition, just as Q-learning replaced the optimality backup by a sample. The critic estimates the value of the current

policy and the actor improves the policy against that estimate, so actor-critic is an approximate form of the generalized policy iteration of Section 14.2. Because the regression target contains $\hat{V}$ itself, the gradient is taken through the prediction $\hat{V}(s_t)$ and not through the target. Such an update is called a *semi-gradient* step: its expected update is not the gradient of any fixed objective.

15.1.1.4 The Chain Example

How much variance does the bootstrap remove, and how much bias does it add? A small chain answers both in closed form. Take ten states $0, 1, \dots, 9$. From state s the agent always moves to $s+1$, and the transition out of state 9 ends the episode and pays 1. Every reward carries independent Gaussian noise of standard deviation $\sigma = 0.15$, and $\gamma = 0.97$. The critic is wrong by $e(s) = 0.5\,(1 - s/9)^2$: badly at the start, not at all at the end. TD learning from a zero initialization tends to produce errors of this shape, because the reward at the end reaches the late states first.

A target that samples n rewards and then bootstraps from the critic at state n inherits the critic's error there, discounted, and the noise of n rewards. Its bias and variance are

$$\text{bias} = \gamma^n e(n), \qquad \text{variance} = \sigma^2 \, \frac{1 - \gamma^{2n}}{1 - \gamma^2}, \tag{15.3}$$

for $n \le 9$; the Monte Carlo target, $n = 10$, has no bias. At $n = 1$ the bias is 0.38 and the variance 0.02; at $n = 10$ the bias is zero and the variance 0.17. The mean squared error, bias squared plus variance, falls from 0.17 at $n = 1$ to 0.10 at $n = 4$ and rises back to 0.17 at $n = 10$.

The best depth lies in the interior because deeper targets bootstrap from better-known states. A critic wrong by the same 0.5 at every state would make the error monotone over the bootstrapped depths (exercise 2). Figure 15.1 draws the family of targets and the measured curves.

15.1.2 The Actor-Critic Update

15.1.2.1 Two Updates from One Number

Both learners now update from the same scalar. Write w for the critic's parameters and $\hat{V}_w$ for the critic. After the transition (s_t, a_t, r_t, s_{t+1}) the two updates are

$$w \leftarrow w + \alpha \, \delta_t \, \nabla_w \hat{V}_w(s_t), \qquad \theta \leftarrow \theta + \eta \, \delta_t \, \nabla_\theta \log \pi_\theta(a_t \mid s_t), \tag{15.4}$$

with step size α for the critic, as for TD learning in Section 14.3, and η for the actor, as for the policy gradient in Section 14.6. The critic's update is the semi-gradient step toward the one-step target. The actor's update is the policy-gradient step of Section 14.7 with δ_t as the weight. One number does both jobs: δ_t tells the critic how wrong its prediction was, and it tells the actor whether the action just taken turned out better or worse than the critic expected (Figure 15.2). Because δ_t needs a single transition, Equation 15.4 can be applied at every step while the agent acts, the way Q-learning updated its table, and nothing has to terminate before learning starts.

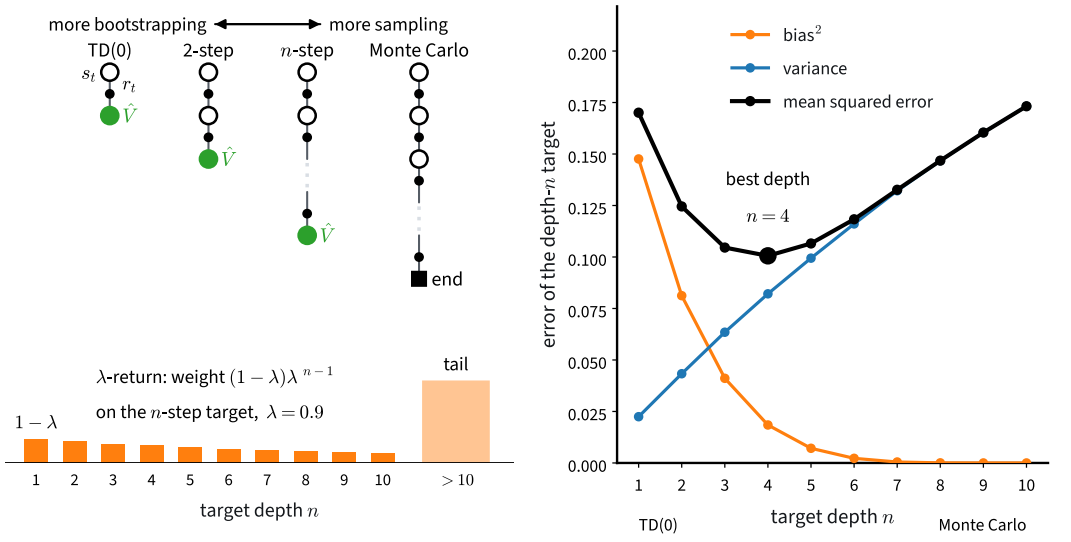
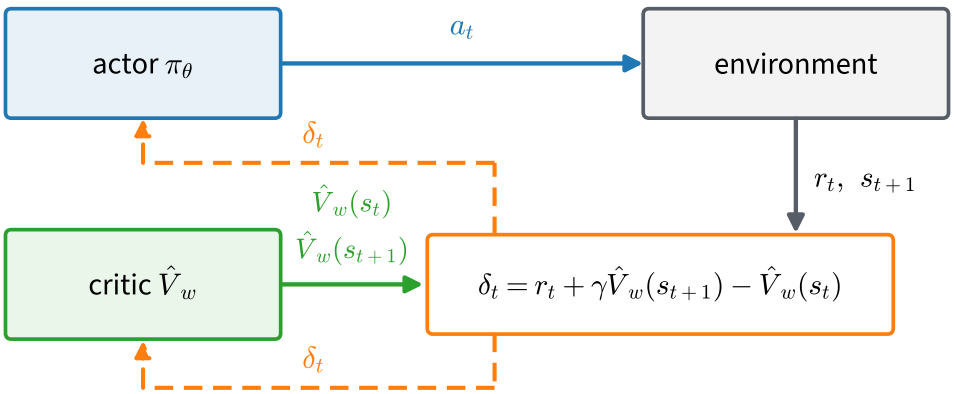


Figure 15.1: The family of bootstrapped targets. Left: backup diagrams of depth one, two, n and to termination, where the green node marks the state at which a target stops sampling and trusts the critic, with the weights $(1-\lambda)\lambda^{n-1}$ that the λ-return of this section puts on them at $\lambda = 0.9$; right: bias, variance and mean squared error of the depth-n target on the ten-state chain of the text, measured over $20{,}000$ rollouts, with the smallest error at $n = 4$.



actor: $\theta \leftarrow \theta + \eta\, \delta_t\, \nabla_\theta \log \pi_\theta(a_t \mid s_t)$ critic: $w \leftarrow w + \alpha\, \delta_t\, \nabla_w \hat{V}_w(s_t)$

the critic stays ahead of the actor: several critic passes per batch, one actor step

Figure 15.2: The actor-critic loop: one transition produces one TD error, which corrects the critic's prediction and weights the actor's step.

The critic's target moves with both networks. An actor step changes the values the critic should predict, and a critic step changes its own targets. If the actor moves faster than the critic can follow, the actor is weighted by the errors of a stale critic. Konda and Tsitsiklis (2000) proved that actor-critic converges, for linear critics and decaying step sizes, when the critic's step sizes shrink more slowly than the actor's, so that the critic keeps up with the current policy. The loop below gets the same effect by giving the critic several regression passes per batch.

15.1.2.2 The Batched Loop

The code does not apply Equation 15.4 after every transition. It keeps the eight-episode batches of Section 14.8 and changes what happens to each batch. The critic first takes `critic_steps` regression passes toward the batch's one-step targets, recomputing the targets from the current critic before every pass. Regression toward targets that contain the regressor's own predictions is called *fitted TD*, and recomputing the targets makes them a moving target. The actor then takes one step weighted by the TD errors of the refitted critic. We call this loop a batched one-step actor-critic. Its best-known relative, A2C (Mnih et al. 2016), weights the same score by n-step advantages, adds an entropy bonus, and takes one joint gradient step per rollout collected from many environments in parallel.

The actor's weight is the raw TD error, not normalized within the batch as the Monte Carlo weight of Section 14.8 was, because of the way Adam scales its steps. Adam divides each gradient coordinate by the square root of a running average of its past squares, an average that with the default decay remembers about a thousand steps, longer than our 150 updates. Its first steps therefore have about the same size whatever the scale of the weights, and later steps shrink only when the gradients shrink below their early size. As the critic improves, the TD errors fall from a standard deviation of about 3 early in training to 0.1 or less near the ceiling, by then mostly the critic's own noise, and Adam's steps fall with them, about eightfold over a run. Normalized TD errors would keep the gradient near its early size, and Adam would keep taking large steps in directions set by that noise (exercise 6).

Early in training the raw weights, and with them the actor's gradient, are larger than a normalized weight would give. We cap the gradient's norm at 0.5, a common value. Under Adam the cap does not limit the size of a step; it limits how much these large early batches count in Adam's running averages. `policy_step_clip` is the `policy_step` of Section 14.5, one gradient step on the weighted log-probabilities of the batch's actions, with the cap added; it returns the norm measured before the cap.

```
def actor_loss(ac, obs, act, weight):
    """-(1/n) sum_t w_t log pi(a_t|s_t), the loss of d2l.policy_step."""
    w, obs, act = (torch.as_tensor(x) for x in (weight, obs, act))
    return -(w * ac.log_prob(obs, act)).mean()

def policy_step_clip(ac, obs, act, weight, clip=grad_clip):
    """d2l.policy_step with the actor's gradient norm capped at `clip`;
    returns the norm measured before the cap."""
    loss = actor_loss(ac, obs, act, weight)
    ac.opt_pi.zero_grad()
    loss.backward()
    gnorm = nn.utils.clip_grad_norm_(ac.policy.parameters(), clip)
    ac.opt_pi.step()
```

```
    return float(gnorm)
```

The loss alone is `actor_loss`, which the probe at the end of the section reuses.

`train` is the loop. Each update collects a batch with `d2l.rollout`, asks a function `weight(ac, b)` for the actor's weight at every step of the batch `b`, takes one capped actor step with these weights, and yields the number of environment steps taken so far, the batch's mean return, and the actor's gradient norm before the cap. The weight function also fits the critic, so the loop stays the same for every estimator of this section and only the weight changes. The actor-critic weight is `td_weight`: `d2l.fit_value` (Section 14.8) takes one Adam step on the squared error between the critic's predictions and a target vector, `ac.value_np` evaluates the critic on a batch of states, and after the critic's passes the weight is the raw TD error of the refitted critic.

```
def train(seed, ac, weight, num_updates=num_updates,
          batch_episodes=batch_episodes):
    """Batched policy gradient with the per-step weights weight(ac, b);
    yields (env steps, mean return of the batch, actor grad norm)."""
    rng, env = np.random.default_rng(seed), gym.make('CartPole-v1')
    env.reset(seed=seed)
    steps = 0
    for _ in range(num_updates):
        b = d2l.rollout(env, ac.act, batch_episodes, rng)
        gnorm = policy_step_clip(ac, b.obs, b.act, weight(ac, b))
        steps += len(b)
        yield steps, float(b.episode_returns().mean()), gnorm

def td_weight(ac, b):
    """Actor-critic: refit the critic toward its one-step target, then
    weight each step by the raw TD error of the refitted critic."""
    for _ in range(critic_steps):   # fresh target, one pass, repeat
        d2l.fit_value(ac, b.obs, b.td_target(ac.value_np, gamma))
    return b.td_target(ac.value_np, gamma) - ac.value_np(b.obs)
```

Fitting the critic before computing the actor's weights is a choice. A2C and most GAE implementations compute the TD errors with the critic that was current when the batch was collected, and update both networks afterward. With one critic step per rollout, as in A2C, the order hardly matters. With twenty passes per batch it can: the refitted critic is much closer to V^π on the batch's own states, so the actor's weights, which here rest entirely on the critic, carry less of its error (exercise 3 compares the two orders).

Fresh batches help the critic in a second way. Each batch comes from the current policy and from the states it visits, so a critic error that matters to the current behavior is likely to meet relevant transitions in the next batch. This feedback does not guarantee convergence: semi-gradient TD with a nonlinear critic can diverge even on its own policy's data (Tsitsiklis and Van Roy 1997). Trained on replayed transitions of older policies, the same update combines function approximation, bootstrapping and off-policy data, the *deadly triad* of Sutton and Barto (2018), and divergence becomes common; Section 15.4 takes up that case.

15.1.2.3 Bootstrapped and Monte Carlo Weights on CartPole

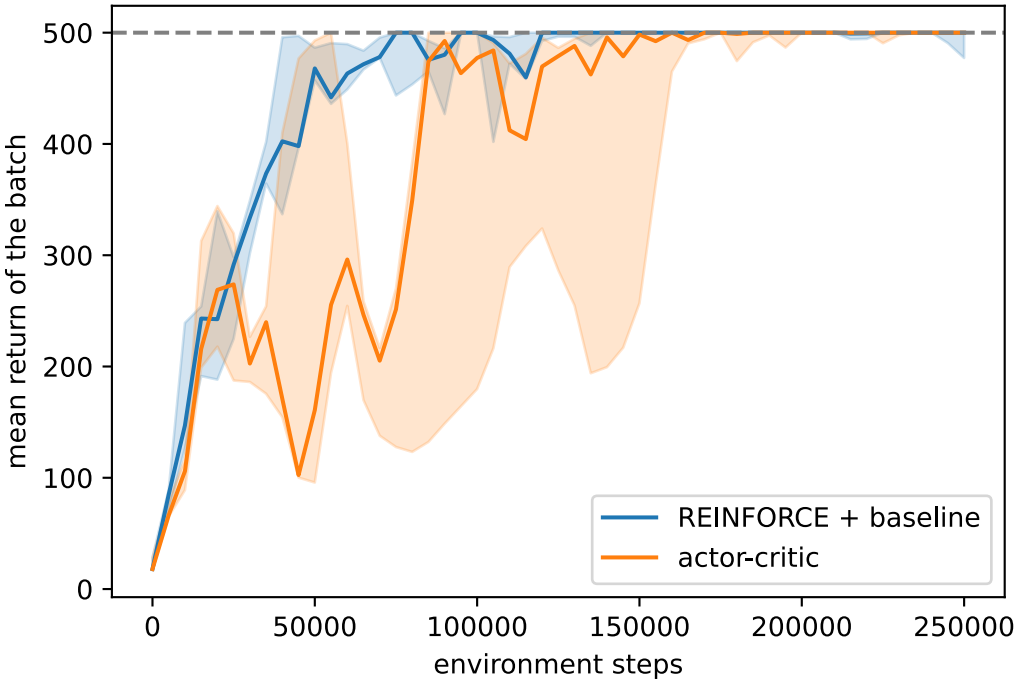
Does the bootstrapped weight learn faster than the Monte Carlo weight it replaces? The reference is REINFORCE with a learned baseline, the loop of Section 14.8, written as a second weight function, `mc_weight`, so that the two arms share every other line. Two more details keep only the weight different: the actor's step goes through `policy_step_clip`, and the critic takes the same `critic_steps` regression passes per batch, here toward the fixed Monte Carlo targets $\widehat{G}_t$. As in Section 14.8, `truncation_bootstrap` adds the critic's discounted value at a time-limit cut to the reward-to-go, `d2l.standardize` standardizes the weights within the batch, and the weights are computed before the critic's passes; the actor and the critic are separate networks, so this order does not change the actor's step.

```
def mc_weight(ac, b):
    """REINFORCE with a learned baseline: the standardized Monte Carlo
    advantage G - V, after which the critic fits G."""
    G = (b.reward_to_go(gamma)
         + b.truncation_bootstrap(gamma, ac.value_np))   # time limits
    w = d2l.standardize(G - ac.value_np(b.obs))
    d2l.fit_value(ac, b.obs, G, num_steps=critic_steps)
    return w
```

We train three seeds of each and plot the batch's mean return against the number of environment steps, the measure of cost that the rest of the chapter uses. `d2l.plot_curves` (the online appendix "Utility Functions and Classes") draws the median over seeds with a band from the lowest to the highest seed. Seeds take different numbers of steps per update, so `d2l.on_grid` first interpolates every run onto a common grid of steps.

```
runs = {'REINFORCE + baseline': np.array(
    [list(train(s, cartpole_agent(s), mc_weight)) for s in range(num_seeds)])}
```

```
runs['actor-critic'] = np.array(
    [list(train(s, cartpole_agent(s), td_weight)) for s in range(num_seeds)])
grid = np.arange(0, 250_001, 5_000)
d2l.plot_curves({name: d2l.on_grid(r, grid, 1) for name, r in runs.items()},
                xlabel='environment steps', ylabel='mean return of the batch',
                x=grid, reference=500)
```



To read the plot by seed, we count the steps each run takes until the mean return of its last ten batches first reaches 450, and how often the cap on the actor's gradient was active. `steps_to` finds the first such step in a run's rows; Section 15.5 uses it again.

```
def steps_to(run, level, window):
    """Env steps at which the mean return of the last `window` rows of
    run = [(env step, return, ...), ...] first reaches `level`; inf if it
    never does."""
    mean = np.convolve(run[:, 1], np.ones(window) / window, 'valid')
    hit = np.flatnonzero(mean >= level)
    return run[hit[0] + window - 1, 0] if len(hit) else np.inf
```

```
for name, r in runs.items():
    k = [f'{steps_to(run, 450, 10) / 1000:.0f}' for run in r]
    print(f'{name:>20}: 450 after {", ".join(k)} thousand steps; '
          f'cap active {(r[:, :, 2] > grad_clip).mean():.0%}')
```

```
REINFORCE + baseline: 450 after 79, 64, 77 thousand steps; cap active 0%
       actor-critic: 450 after 113, 187, 112 thousand steps; cap active 29%
```

The Monte Carlo weight reaches the ceiling first on at least two of the three seeds shown. Three seeds are a small sample; over 40 seeds, with the same twenty critic passes per batch, REINFORCE reached 450 first on at least 35, typically with roughly a sixth to a third fewer environment steps, and no REINFORCE run ended far below the ceiling, while two actor-critic runs did. The cap is active on a fifth to two fifths of actor-critic's updates and on none of REINFORCE's, whose normalized weights keep the gradient small.

On CartPole the bootstrap does not even make the runs steadier. The reward is a determin-

istic $+1$ per step, and the bootstrap at the time limit already replaces the long tail of every cut episode by the critic's value, so the Monte Carlo weight is far less noisy than its worst case. What remains is the critic's bias, and it is largest while the critic is young, the phase in which actor-critic falls behind. The uniform $\delta_t \approx 1$ of an untrained critic is the extreme case.

15.1.2.4 The Two Weights on Shared Batches

If the bootstrap does not speed up learning here, what does it change? We measure both weights on the same batches. The cell below reruns the actor-critic arm for one seed with a weight that calls `td_weight` and, at every update, also records the standard deviation of the TD error and of the Monte Carlo advantage `A_mc`, the weight of `mc_weight` before standardization.

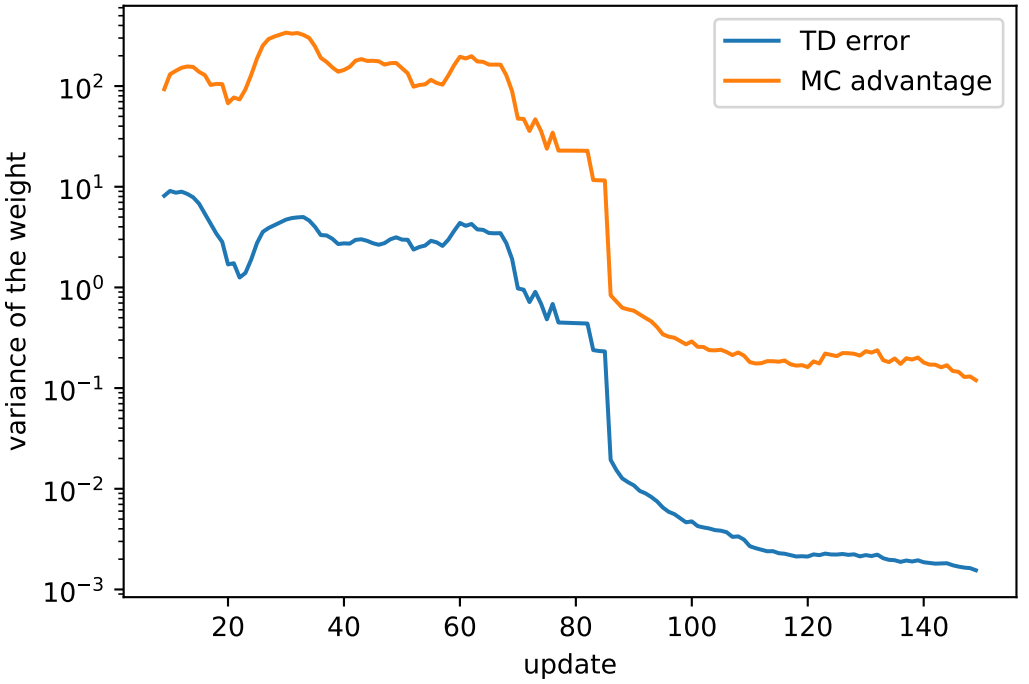
```
trace = []   # per update: std of the TD error, std of the MC advantage

def traced_td_weight(ac, b):
    """td_weight, recording the spread of both weights on its batch."""
    delta = td_weight(ac, b)
    A_mc = (b.reward_to_go(gamma)   # G of mc_weight, minus V
            + b.truncation_bootstrap(gamma, ac.value_np) - ac.value_np(b.obs))
    trace.append((delta.std(), A_mc.std()))
    return delta

for _ in train(0, cartpole_agent(0), traced_td_weight):
    pass
trace = np.array(trace)
```

```
for name, rows in (('first 10', trace[:10]), ('last 50', trace[-50:])):
    print(f'{name:>8} updates: std of the TD error {np.median(rows[:, 0]):.2f}'
          f', of the MC advantage {np.median(rows[:, 1]):.2f}')
print(f'median variance ratio, MC over TD: '
      f'{np.median((trace[:, 1] / trace[:, 0]) ** 2):.0f}x')
d2l.plot_curves({'TD error': trace[:, 0] ** 2, 'MC advantage': trace[:, 1] ** 2},
                xlabel='update', ylabel='variance of the weight', smooth=10,
                yscale='log')
```

```
first 10 updates: std of the TD error 2.69, of the MC advantage 7.69
 last 50 updates: std of the TD error 0.04, of the MC advantage 0.41
median variance ratio, MC over TD: 56x
```



On the same transitions, the Monte Carlo advantage has two to three times the TD error's standard deviation in the first updates, a variance five to ten times larger, and over the whole run its variance is a few tens of times the TD error's; the plot's scale is logarithmic. The TD error depends on one reward and one transition; the Monte Carlo advantage depends on every reward and transition left in the episode. Both variances drop sharply once nearly every episode reaches the 500-step cut: the TD error's because the critic is then accurate, the Monte Carlo advantage's because the bootstrap replaces the tail of every cut episode by the critic's value.

The variance saving is real, but on CartPole it is not what limits learning. REINFORCE averages eight episodes per update and normalizes its weights, and with a well-fitted baseline that suffices; the critic's bias, by contrast, slows every update until the critic catches up.

The bootstrap's lasting advantage is structural: the TD error needs one transition, so an update can come before the episode ends. In a continuing task the return never arrives, and in a task whose episodes last thousands of steps, eight episodes per update take tens of thousands of steps. There a critic is the only option (exercise 5 builds the fully online form). Between these extremes lies a dial: bootstrap after several steps instead of one, and keep most of the variance saving while taking on less of the critic's bias.

15.1.3 Multi-Step Returns and GAE

15.1.3.1 n-Step Returns

The one-step target and the Monte Carlo return are the endpoints of a family. Include n sampled rewards and then bootstrap from the critic:

$$\hat{G}_t^{(n)} = r_t + \gamma\, r_{t+1} + \cdots + \gamma^{n-1} r_{t+n-1} + \gamma^n\, \hat{V}(s_{t+n}), \tag{15.5}$$

and use $\hat{G}_t^{(n)} - \hat{V}(s_t)$ as the advantage estimate. For $n = 1$ this is δ_t; when n reaches the end of the episode, it is the Monte Carlo advantage. Each additional step discounts the critic's error by another factor γ and evaluates it at a later state, and adds another sampled reward and transition. This is the trade of the chain example, where $n = 4$ balanced the two.

15.1.3.2 The λ-Return and the Telescoping Identity

Rather than pick one depth, we can average them all, with geometric weights that make the average computable:

$$\hat{A}_t^{\text{GAE}} = (1-\lambda)\sum_{n=1}^{\infty} \lambda^{n-1}\big(\hat{G}_t^{(n)} - \hat{V}(s_t)\big), \qquad \lambda \in [0,1]. \tag{15.6}$$

The weights $(1-\lambda)\lambda^{n-1}$ sum to one. At $\lambda = 0$ all weight is on the one-step estimate. On an episode that terminates, every depth past the end equals the Monte Carlo advantage, so as $\lambda \to 1$ the mixture tends to the Monte Carlo advantage. Written this way, the estimator seems to need every depth computed separately. It does not, because the mixture collapses.

Proposition. The mixture is a discounted sum of TD errors:

$$\hat{A}_t^{\text{GAE}} = \sum_{l=0}^{\infty} (\gamma\lambda)^l \, \delta_{t+l}. \tag{15.7}$$

Proof. Each depth telescopes into TD errors: $\sum_{l=0}^{n-1} \gamma^l \delta_{t+l} = \sum_{l=0}^{n-1} \gamma^l r_{t+l} + \gamma^n \hat{V}(s_{t+n}) - \hat{V}(s_t) = \hat{G}_t^{(n)} - \hat{V}(s_t)$, the interior $\hat{V}$ terms cancelling in pairs. Substitute this into Equation 15.6 and swap the two sums: the pair (n, l) with $l < n$ contributes $(1-\lambda)\lambda^{n-1}\gamma^l\delta_{t+l}$, so $\hat{A}_t^{\text{GAE}} = \sum_{l\ge 0} \gamma^l \delta_{t+l} \, (1-\lambda) \sum_{n \ge l+1} \lambda^{n-1}$. The inner geometric sum is λ^l. ■

The identity turns an average over all depths into one backward pass over the episode, linear in its length, which is how implementations compute it (Schulman et al. 2016). On an episodic task the sums stop at termination, where $\hat{V} = 0$ and no rewards follow, so every δ past the end is zero and the infinite sum is also the finite one.

15.1.3.3 GAE as a Backward Scan

The sum Equation 15.7 is an algorithm for *generalized advantage estimation* (GAE): compute the TD errors, then accumulate them backward with factor $\gamma\lambda$, restarting at every episode boundary. The method `Batch.backward_scan` of Section 14.7, which computes the reward-to-go by the recurrence $\hat{G}_t = r_t + \gamma \hat{G}_{t+1}$, already implements this recurrence; only its input and its factor change.

```
@d2l.add_to_class(d2l.Batch)
def gae(self, value_fn, gamma, lam):
    """GAE(gamma, lam): the reward-to-go scan, run on the TD errors."""
    delta = self.td_target(value_fn, gamma) - value_fn(self.obs)
    return self.backward_scan(delta, gamma * lam)
```

Both endpoints are identities that we can check on a batch. At $\lambda = 0$ the scan returns the TD errors themselves. At $\lambda = 1$, on episodes that terminate, the telescoping runs to termination and returns the reward-to-go minus the baseline, the Monte Carlo advantage of Section 14.7.

```
env = gym.make('CartPole-v1')
env.reset(seed=0)
fresh = cartpole_agent(0)
b = d2l.rollout(env, fresh.act, 8, np.random.default_rng(0))
V = fresh.value_np(b.obs)
delta = b.td_target(fresh.value_np, gamma) - V
assert np.allclose(b.gae(fresh.value_np, gamma, 0.0), delta)
assert np.allclose(b.gae(fresh.value_np, gamma, 1.0),
                   b.reward_to_go(gamma) - V, atol=1e-4)
print('lambda = 0 is the TD error; lambda = 1 is the Monte Carlo advantage')
```

```
lambda = 0 is the TD error; lambda = 1 is the Monte Carlo advantage
```

The $\lambda = 1$ identity needs episodes that terminate. On an episode cut off by the time limit, `gae` bootstraps the missing future through its last TD error, while `reward_to_go` treats the future as empty until `truncation_bootstrap` adds it back; the untrained policy above drops the pole long before the cut, so the assertion holds. The same backward sum has an online form, TD(λ), which keeps a decaying *eligibility trace* per critic parameter and suits agents that learn from a stream without storing episodes (Sutton 1988; Elsayed et al. 2024).

15.1.4 Bias and Variance of the λ-Return

15.1.4.1 The λ Sweep

Which λ trains best? The dial needs a new weight and a new critic target. The actor is weighted by the GAE advantage, and the critic is fitted toward the λ-return, the advantage plus the value it is measured from, which `lambda_return` computes; at $\lambda = 0$ it is the one-step target of `td_weight`. Section 15.2 fits its critic the same way.

```
@d2l.add_to_class(d2l.Batch)
def lambda_return(self, value_fn, gamma, lam):
    """The critic's target for GAE: the advantage plus the value it is
    measured from."""
    return self.gae(value_fn, gamma, lam) + value_fn(self.obs)
```

`gae_weight(lam)` returns the weight function of one setting of the dial, and `gae_run` trains one seed with it. Unlike `td_weight`, the weight is standardized, because near $\lambda = 1$ it has the scale of the Monte Carlo advantage, with a variance of several tens, and raw weights would hit the cap at large λ and not at small λ. The budget is short, 50 updates of four episodes, so the comparison shows which estimator helps an agent whose critic is still being learned.

```
lams = [0.0, 0.5, 0.9, 0.95, 1.0]

def gae_weight(lam):
    """The weight of GAE(gamma, lam), standardized so that every lam steps
```

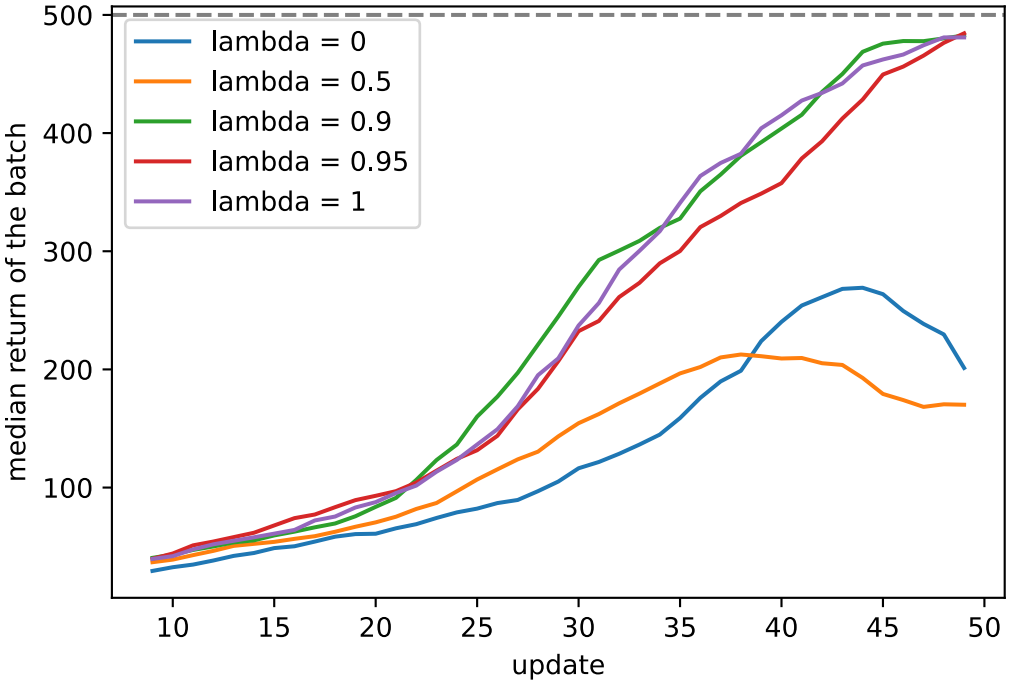
```
    alike; the critic first fits the lambda-return."""
    def weight(ac, b):
        for _ in range(critic_steps):
            d2l.fit_value(ac, b.obs, b.lambda_return(ac.value_np, gamma, lam))
        return d2l.standardize(b.gae(ac.value_np, gamma, lam))
    return weight

def gae_run(seed, lam):
    """The returns of one short run: 50 updates of four episodes."""
    return [ret for _, ret, _ in train(seed, cartpole_agent(seed),
                                       gae_weight(lam), num_updates=50,
                                       batch_episodes=4)]

sweep = {lam: d2l.run_seeds(gae_run, 5, lam=lam) for lam in lams}
```

```
for lam, r in sweep.items():
    best = np.array([np.convolve(c, np.ones(20) / 20, 'valid').max()
                     for c in r])
    print(f'lambda = {lam:4}: best 20-update window, '
          f'median {np.median(best):5.1f}, seeds {np.round(np.sort(best))}')
d2l.plot_curves({f'lambda = {lam:g}': np.median(r, axis=0)
                 for lam, r in sweep.items()},
                xlabel='update', ylabel='median return of the batch',
                smooth=10, reference=500)
```

```
lambda =  0.0: best 20-update window, median 222.5, seeds [138. 198. 223. 283. 328.]
lambda =  0.5: best 20-update window, median 203.3, seeds [175. 179. 203. 218. 321.]
lambda =  0.9: best 20-update window, median 419.5, seeds [387. 400. 420. 454. 486.]
lambda = 0.95: best 20-update window, median 406.9, seeds [327. 367. 407. 444. 482.]
lambda =  1.0: best 20-update window, median 449.7, seeds [293. 385. 450. 464. 479.]
```



In the median over seeds, $\lambda = 0$ and $\lambda = 0.5$ reach a best 20-update window of about 150 to 250, far below the ceiling, while $\lambda \geq 0.9$ reaches about twice as high. Individual seeds overlap across this divide, so we compare medians. On a short budget the critic is wrong for most of training, and the estimators that lean on it least learn fastest. This is the CartPole comparison above seen from the other end: on CartPole a young critic's bias costs more than the variance it removes.

15.1.4.2 A Frozen-Policy Probe

Training curves mix the quality of the estimator with optimization and changing data. To examine the estimator alone, we freeze a partly trained agent, draw many batches at its fixed parameters, and compare the policy-gradient estimate of each λ with a reference. The reference is the mean of the $\lambda = 1$ estimates over all draws. When the episodes terminate before the time limit, this approximates the unbiased Monte Carlo gradient without a model of the environment (exercise 4 treats truncated episodes). `grad_vec` returns the gradient of one batch's weighted log-probabilities as a flat vector, and `freeze` trains a fresh agent for 40 actor-critic updates, about a quarter of a full run, when its critic is still wrong.

```
def grad_vec(ac, b, w):
    """The actor's gradient on batch b with weights w, as one flat vector."""
    g = torch.autograd.grad(actor_loss(ac, b.obs, b.act, w),
                            list(ac.policy.parameters()))
    return np.concatenate([x.numpy().ravel() for x in g])
```

```
def freeze(seed, warmup=40):
    """An agent after `warmup` actor-critic updates: its critic still
    wrong."""
    ac = cartpole_agent(seed)
    for _ in train(seed, ac, td_weight, num_updates=warmup):
        pass
    return ac
```

`probe` draws 200 batches of four episodes at a frozen agent and returns, for every λ, the bias of the mean estimate and the variance of a single estimate, the bias relative to the reference's length and the variance relative to its squared length. Their sum, bias squared plus variance, is the expected squared error of one draw.

```
def probe(ac, seed, num_draws=200):
    """Relative bias and variance of the batch gradient for every lambda,
    against the mean of the lambda = 1 estimates."""
    rng, env = np.random.default_rng(seed), gym.make('CartPole-v1')
    env.reset(seed=seed)
    draws = {lam: [] for lam in lams}
    for _ in range(num_draws):
        b = d2l.rollout(env, ac.act, 4, rng)
        for lam in lams:
            draws[lam].append(grad_vec(ac, b, b.gae(ac.value_np, gamma, lam)))
    u = {lam: np.stack(d) for lam, d in draws.items()}
    u_ref = u[1.0].mean(axis=0)   # the many-rollout reference
    bias = [np.linalg.norm(u[l].mean(0) - u_ref) for l in lams]
    var = [((u[l] - u[l].mean(0)) ** 2).sum(1).mean() for l in lams]
    return (np.array(bias) / np.linalg.norm(u_ref),
            np.array(var) / (u_ref ** 2).sum())
```

One frozen agent is a single draw from many possible ones, and the best λ for one agent depends on how wrong its particular critic is. We therefore freeze eight agents and report, per λ, the median over agents of the bias, of the variance, and of the error divided by the same agent's error at $\lambda = 1$:

```
stats = np.array([probe(freeze(s), s) for s in range(8)])  # agent, stat, lam
err = stats[:, 0] ** 2 + stats[:, 1]
ratio = err / err[:, -1:]
for i, lam in enumerate(lams):
    print(f'lambda = {lam:4}: bias {np.median(stats[:, 0, i]):4.2f}, '
          f'variance {np.median(stats[:, 1, i]):4.2f}, '
          f'error / error at 1: {np.median(ratio[:, i]):4.2f}')
hits = np.isin(np.array(lams)[err.argmin(axis=1)], [0.9, 0.95]).sum()
print(f'{hits} of {len(stats)} agents have their smallest error at lambda = 0.9 '
      f'or 0.95')
```

```
lambda =  0.0: bias 1.19, variance 0.04, error / error at 1: 1.91
lambda =  0.5: bias 1.13, variance 0.05, error / error at 1: 1.61
lambda =  0.9: bias 0.84, variance 0.17, error / error at 1: 0.83
lambda = 0.95: bias 0.53, variance 0.32, error / error at 1: 0.61
lambda =  1.0: bias 0.00, variance 0.98, error / error at 1: 1.00
```

```
7 of 8 agents have their smallest error at lambda = 0.9 or 0.95
```

Across the eight frozen agents, the median error is smallest at $\lambda = 0.9$ or 0.95, about half the error at $\lambda = 1$ and below the error at $\lambda = 0$. From $\lambda = 0$ to $\lambda = 1$ the bias falls to zero and the variance rises more than tenfold, most of the rise in the last step, where the weight $(\gamma\lambda)^l$ of distant TD errors stops decaying. Individual agents vary: most have their smallest error at 0.9 or 0.95, and in the others a smaller λ wins, as the proposition below predicts for a critic that is already accurate. The $\lambda = 1$ row has zero bias by construction, because its own mean is the reference. The common default in PPO implementations is $\lambda = 0.95$ (Schulman et al. 2017).

15.1.4.3 The Source of the Interior Optimum

Why should the best λ lie strictly between the endpoints? A perfect critic gives the answer by contrast.

Proposition. Let the critic be exact, $\hat{V} = V^\pi$, and let episodes have no time limit. Then every λ gives an unbiased estimate of $A^\pi(s_t, a_t)$, and

$$\mathbb{E}\big[(\hat{A}_t^{\mathrm{GAE}})^2 \mid s_t, a_t\big] = \sum_{l \geq 0} (\gamma\lambda)^{2l}\, \mathbb{E}\big[\delta_{t+l}^2 \mid s_t, a_t\big],$$

so the error of the estimate is smallest at $\lambda = 0$.

Proof. With an exact critic the TD error at any state has mean zero given that state: $\mathbb{E}[\delta_{t+l} \mid s_{t+l}] = \sum_a \pi(a \mid s_{t+l})\, Q^\pi(s_{t+l}, a) - V^\pi(s_{t+l}) = 0$, by the identity of the previous section averaged over the action. By the Markov property this holds also given the whole history up to s_{t+l}, which for $l \geq 1$ includes s_t, a_t and every earlier TD error. So every term with $l \geq 1$ in Equation 15.7 has mean zero given (s_t, a_t), and the estimate's mean is $\mathbb{E}[\delta_t \mid s_t, a_t] = A^\pi(s_t, a_t)$. For $k < l$ the cross term $\mathbb{E}[\delta_{t+k}\, \delta_{t+l} \mid s_t, a_t]$ vanishes for the same reason, leaving the sum of squares. Each squared term is nonnegative and its weight $(\gamma\lambda)^{2l}$ grows with λ, while the mean does not change, so the mean squared error grows with λ. ■

With an exact critic, the sampled tail only adds noise, and the one-step TD error is best. An interior optimum therefore measures the critic's error: it exists because the critic is wrong, and it should move toward $\lambda = 0$ as the critic improves, which exercise 8 tests at a frozen policy. The default $\lambda = 0.95$ is in this sense a statement about how accurate critics usually are during training.

15.1.4.4 A Critic or a Group Baseline

A learned critic is one way to estimate the advantage. Section 14.7 met another, the group baseline, which compares several samples from the same start state with one another. Which one to use follows from the trade of this section. A critic is the better estimator where rewards arrive at every step, episodes are long, and states are cheap to evaluate: its error shrinks as it learns, and it lets an update come before the episode ends. A group baseline is the better estimator where each episode ends in a single reward and samples are expensive, as when a language model answers a prompt and a checker grades the answer. A critic would have to predict the final grade from every prefix of every response, a hard regression problem of its own, while the other responses to the same prompt give a baseline with no second network. GRPO, whose weights Section 14.7 derived, makes this choice.

Deployed actor-critic implementations also do not collect whole episodes. They step many environments in parallel for a fixed number of steps, cut every segment there whether or not its episodes have ended, and update. The cut is handled like a time limit: the last TD error of an unfinished segment bootstraps from the critic's value at the cut, which is what `td_target` does on a truncated episode, and the backward scan of `gae` restarts at every cut. Fixed-length segments make the interval between updates independent of the episode length, the structural property of the bootstrap that whole-episode batches leave unused.

We can now update a policy from a single transition and choose how much of the sampled tail to keep: the TD error for one step, the λ-return for a geometric mix of all depths, and GAE to compute the mix in one backward pass. On CartPole the best choice keeps most of the tail, λ near 0.95, because the critic is young for much of training. Two limits of Section 14.8 remain: every batch is used for one actor step and then discarded, and nothing bounds how far that step moves the policy. Section 15.2 takes up both.

15.1.5 Summary

An actor-critic method weights the policy gradient by the TD error of a learned critic instead of the sampled return. With an exact critic the TD error is an unbiased one-transition sample of the advantage; with a learned one it trades the variance of the sampled tail for the critic's bias, and the critic's own update is a semi-gradient step of sampled policy evaluation. On CartPole, with equal critic budgets, the Monte Carlo weight reaches the ceiling in fewer environment steps, because the young critic's bias does more harm than the variance it removes; the bootstrap's lasting advantage is that an update needs no finished episode. The n-step returns and the λ-return interpolate between the two weights, and GAE computes the λ-return as a backward scan of TD errors. At a frozen policy the error is smallest near $\lambda = 0.9$ to 0.95; with an exact critic it would be smallest at $\lambda = 0$, so the interior optimum measures the critic's error.

15.1.6 Exercises

1. **Bias of the bootstrapped weight.** The conditional zero-mean identity of Section 14.7 allows any function $b(s_t)$ of the current state to be subtracted from the weight of the score $\psi_t = \nabla_\theta \log \pi_\theta(a_t \mid s_t)$ without biasing the gradient. Write $e(s) = \hat{V}(s) - V^\pi(s)$ for the critic's error, with $e = 0$ at terminal states, where both values are zero.
   1. Show that the TD error of Equation 15.2 splits as $\delta_t = \delta_t^\pi + \gamma\, e(s_{t+1}) - e(s_t)$, where δ_t^π is the TD error of the exact critic V^π. The identity removes one of the two error terms. For the other one, which step of the identity's proof fails?
   2. Show that the bias of the step-t term $\mathbb{E}[\delta_t\, \psi_t]$ relative to $\mathbb{E}[A^\pi(s_t, a_t)\, \psi_t]$ is $\gamma\, \mathbb{E}\big[\bar{e}(s_t, a_t)\, \psi_t\big]$ with $\bar{e}(s, a) = \mathbb{E}[e(s_{t+1}) \mid s_t = s, a_t = a]$. Under what condition on e does the bias vanish? Is $\hat{V} = V^\pi$ necessary for it?
   3. Let the critic overestimate every nonterminal state by the same constant $c > 0$. At a state where one action ends the episode and the other does not, which action does the bias favor, and what changes for $c < 0$? On CartPole, which pays 1 per step, a freshly initialized critic predicts values near zero while V^π is positive. In which direction does this model predict the bias to push the policy early in training?
2. code **Optimal bootstrap depth.** Take the chain example of this section: $\sigma = 0.15$, $\gamma = 0.97$, and the critic error $e(s) = 0.5\,(1 - s/9)^2$.

1. Derive the bias $\gamma^n e(n)$ and the variance $\sigma^2(1-\gamma^{2n})/(1-\gamma^2)$ of the depth-n target from state 0. Then simulate $20{,}000$ rollouts of every depth, compare the measured mean squared errors with the formulas, and confirm the optimum $n = 4$.
2. Replace the critic's error by the constant $e(s) = 0.5$ at every nonterminal state. Show that for $n \le 9$ the mean squared error is

$$\frac{\sigma^2}{1-\gamma^2} + \gamma^{2n}\Big(0.25 - \frac{\sigma^2}{1-\gamma^2}\Big), \tag{15.8}$$

   so that it is monotone over the bootstrapped depths for every σ. Does it remain monotone once the Monte Carlo target $n = 10$ is included? Which depths can be optimal for such a critic, and which one is optimal at $\sigma = 0.15$?
3. Return to the tapered error and find the optimal depth for $\sigma = 0.04$ and for $\sigma = 0.3$. Explain the direction in which the optimum moves.

3. code **Critic passes and the order of the updates.** `td_weight` gives the critic `critic_steps` regression passes per batch and recomputes `td_target` before every pass.
   1. Rerun the actor-critic arm with `critic_steps` in $\{1, 5, 20\}$, two seeds each, and plot the curves. How does the number of passes change the speed of learning and the behavior near the ceiling of 500?
   2. Compute the target once per batch instead, and fit it with `d2l.fit_value(ac, b.obs, target, num_steps=critic_steps)`. Train the same two seeds at 20 passes. Ignoring generalization between states, how many steps back along an episode can a reward's influence on $\hat{V}$ travel during one batch with a fixed target? How many steps can it travel at most with a target recomputed before every pass? Use the two answers to explain the difference between the fixed-target and the recomputed-target curves at 20 passes.
   3. Move the line that computes `delta` above the critic's passes, so that the actor is weighted by the TD errors of the critic that was current when the batch was collected. Train ten seeds under each order and count the runs whose last ten batches average below 300. Why does the order matter more with twenty passes than with one?
4. **The λ-return recursion.** Write $G_t^\lambda = \hat{A}_t^{\mathrm{GAE}} + \hat{V}(s_t)$ for the target on which `gae_weight` fits the critic.
   1. From Equation 15.7, show that $\hat{A}_t^{\mathrm{GAE}} = \delta_t + \gamma\lambda\,\hat{A}_{t+1}^{\mathrm{GAE}}$ within an episode, and conclude that at a nonterminal step that is not the last of its episode

$$G_t^\lambda = r_t + \gamma\big((1-\lambda)\,\hat{V}(s_{t+1}) + \lambda\,G_{t+1}^\lambda\big). \tag{15.9}$$

      What does the recursion become at the last step of an episode that terminates, and at the last step of an episode that the time limit truncates?
   2. Let an episode be truncated after T steps, and let s_T be its last `next_obs`. Show that at step t of this episode `b.gae(value_fn, gamma, 1.0)` exceeds `b.reward_to_go(gamma) - value_fn(b.obs)` by exactly $\gamma^{T-t}\,\hat{V}(s_T)$. For general λ, compute the derivative of G_t^λ with respect to $\hat{V}(s_T)$. Which steps of a truncated episode does the critic's value at the truncation point influence when $\lambda < 1$?
   3. The weight of δ_{t+l} in Equation 15.7 is $(\gamma\lambda)^l$. Derive the half-life τ, the number of steps after which this weight has fallen to half its value at $l = 0$, and evaluate it at $\gamma = 0.99$ for $\lambda \in \{0.5, 0.9, 0.95, 1\}$. The variance that the frozen-policy probe prints grows most between $\lambda = 0.95$ and $\lambda = 1$. Relate this to the half-lives.

Adapted from Richard S. Sutton and Andrew G. Barto, Reinforcement Learning: An Introduction, Exercises 12.1 and 12.2.

5. [extended] **Fully online.** ▣ The update Equation 15.4 can be applied after every transition, with no batch at all: one critic step toward the single target $r_t + \gamma \widehat{V}(s_{t+1})$, masked at termination, and one actor step weighted by δ_t itself, as in `td_weight`.
   1. Implement this variant with `d2l.fit_value` and `policy_step_clip` applied to arrays of length one. Count the environment steps of one actor-critic run of `train` and give the online variant the same number. Train three seeds at the default learning rate of `ActorCritic.mlp` and at one ten times smaller, passed as `lr` inside `cartpole_agent`, and compare the curves with those of the batched loop.
   2. Successive updates now come from successive states of one episode. What goes wrong? A replay buffer stores past transitions and trains on minibatches drawn uniformly from them. Which of the failures would it address? Why can the actor's score-function update not simply train on replayed transitions?
   3. Parallel environments decorrelate the data without a replay buffer (Mnih et al. 2016). Step eight CartPole environments in lockstep and update on each slice of eight transitions, one from each environment. Which of the failures does this repair, and which does it leave?
6. code **Gradient norms and the cap.** ▣ The norm that `policy_step_clip` reports is that of $\hat{\mathbf{g}} = \frac{1}{N}\sum_t \hat{A}_t\, \psi_t$, where $\hat{A}_t$ is the actor's weight, the normalized Monte Carlo advantage of `mc_weight` and the raw TD error of `td_weight`, and $\psi_t = \nabla_\theta \log \pi_\theta(a_t \mid s_t)$. The last column of `runs` records it for every update.
   1. Compare the median norm of the two arms in `runs`. Treat the N terms as independent and identically distributed and show that
   $$\mathbb{E}[\|\hat{\mathbf{g}}\|^2] = \left\|\mathbb{E}[\hat{A}_t\, \psi_t]\right\|^2 + \frac{1}{N}\Big(\mathbb{E}[\|\hat{A}_t\, \psi_t\|^2] - \left\|\mathbb{E}[\hat{A}_t\, \psi_t]\right\|^2\Big). \tag{15.10}$$
   Split the weight into its signal $\mathbb{E}[\hat{A}_t \mid s_t, a_t] - \mathbb{E}[\hat{A}_t \mid s_t]$ and the rest, and show that only the signal contributes to $\mathbb{E}[\hat{A}_t\, \psi_t]$. What does a large remainder of a unit-variance weight do to the norm?
   2. At the agent `freeze(0)`, draw 50 batches of eight episodes with `d2l.rollout`. For each batch, compute the norm of `grad_vec(ac, b, d2l.standardize(w))` for the TD error and for the Monte Carlo advantage `b.reward_to_go(gamma) - ac.value_np(b.obs)`, and again after randomly permuting each weight vector within the batch. Normalizing both weights removes the difference in scale that the raw TD error of `td_weight` carries. What does the permutation remove? How does the norm of each estimator compare with that of its permuted version?
   3. Modify the actor-critic arm to record in every batch the standard deviation of `delta` (in `td_weight`) and the norm of the change in the actor's parameters made by the step (in `train`). Train ten seeds under three rules: as written, with `clip=float('inf')`, and with `d2l.standardize(delta)` as the weight. Compare the parameter change over the first ten and the last fifty updates under each rule, and count the runs that fall below a running mean of 300 after reaching 475. Use the recorded standard deviations and Adam's per-coordinate scaling to explain why normalizing a weight that shrinks toward critic noise can hurt late in training, and what the cap changes under Adam.
7. code **Correlation of the two weights.** ▣ Suppose the critic is exact, $\widehat{V} = V^\pi$, and the episodes have no time limit.
   1. With the identity $\widehat{G}_t - \widehat{V}(s_t) = \delta_t + \gamma(\widehat{G}_{t+1} - \widehat{V}(s_{t+1}))$ and the proposition

on the interior optimum, show that the covariance of δ_t with the Monte Carlo advantage equals the variance of δ_t, so that their correlation is the square root of the ratio of the two variances.

2. Extend the cell that fills `trace` to record also the correlation q of `delta` with `A_mc` in every batch, and write v_{TD} and v_{MC} for the two variances. Compute $\kappa = q\sqrt{v_{\mathrm{MC}}/v_{\mathrm{TD}}} - 1$, the covariance of δ_t with the remaining tail in units of the variance of δ_t, and plot it against the update. It is zero for an exact critic. How does κ behave early and late in training?

8. code **Bias of GAE.** ⊠ Problem 1 found the bias of the weight δ_t in terms of the critic's error $e = \hat{V} - V^{\pi}$. This problem extends the result to every λ; take episodes that terminate, with $e = 0$ at and after termination.

   1. Show that

$$\hat{A}_t^{\mathrm{GAE}} = \sum_{l \geq 0} (\gamma\lambda)^l \, \delta_{t+l}^{\pi} - e(s_t) + (1-\lambda) \sum_{k \geq 1} \gamma^k \lambda^{k-1} \, e(s_{t+k}), \tag{15.11}$$

   where δ^{π} is the TD error of V^{π}. With the proposition on the interior optimum, conclude that the bias of the step-t term of the gradient estimate is $\mathbf{b}(\lambda) = (1-\lambda)\sum_{k \geq 1} \gamma^k \lambda^{k-1} \, \mathbf{c}_k$ with $\mathbf{c}_k = \mathbb{E}[e(s_{t+k})\, \psi_t]$.

   2. Suppose $\mathbf{c}_k = q^{k-1}\mathbf{c}_1$ for a fixed vector $\mathbf{c}_1$ and some $0 \leq q \leq 1$: the action's influence on the critic's error at later states decays geometrically. Show that $\|\mathbf{b}(\lambda)\|/\|\mathbf{b}(0)\| = (1-\lambda)/(1-\gamma\lambda q)$, and evaluate this ratio for the values in `lams` at $\gamma = 0.99$ with $q = 1$ and $q = 0.9$. For each of the eight agents of the frozen-policy probe, divide its bias at each λ by its bias at $\lambda = 0$ and compare the results with the predicted ratios. Which q fits? If a ratio exceeds 1, what does that say about the vectors $\mathbf{c}_k$?

   3. Hold the policy of `freeze(0)` fixed and keep fitting only its critic: run 50 further batches of the critic's passes without the actor's step, and repeat the probe. How do the relative bias and the λ with the smallest one-draw error move as the critic is fitted longer at a fixed policy? Does the measured optimum move toward the $\lambda = 0$ that the proposition predicts for an exact critic?

9. **A critic on a one-step task.** The verifier task of Section 14.1 is an episode of one step: the prompt x is the state, the response y the action, and the episode terminates with reward $r(x, y)$, one for a correct answer and zero otherwise.

   1. Write the TD error of a critic $\hat{V}(x)$ on this task and show that it is the reward minus a per-prompt baseline. Which value of $\hat{V}(x)$ is the exact critic, and how does its TD error compare with the group-centered weight of Section 14.7 for a group of K responses to the same prompt?

   2. What can a learned critic do on this task that the group baseline cannot, and what does it cost? Consider a policy that changes after every update and a critic fitted from the same samples.

   3. Now let the response be generated one token at a time, with the reward only after the last token. With an exact critic of every prefix, which TD errors along a response are nonzero, and what must the critic predict that the group baseline never needs?

15.2 Trust Regions and Proximal Policy Optimization

The agents of Section 14.8 and Section 15.1 collect eight episodes, take one gradient step, and throw the batch away. Once the agent balances the pole well, those eight episodes are four thousand steps of experience, and all of it informs a single update. The step is also unguarded. Nothing stops a large update from moving the policy a long way, and the next batch is then collected by whatever policy that update produced. Can we take several steps on one batch, and how far may they go?

The answer starts with a fact about parameters. The same change of the parameters can leave a policy's behavior almost unchanged or change it completely, depending on where it starts, so a step size in parameter space does not control the quantity that matters, the change in behavior. In 2015 John Schulman and colleagues made that change the thing to control. Trust Region Policy Optimization (TRPO) bounds the divergence between the old and the new action distributions at the visited states, guided by a theorem on how much improvement such a step guarantees (Schulman et al. 2015). Two years later Proximal Policy Optimization (PPO) replaced the constraint by a clipped objective that any gradient optimizer can run (Schulman et al. 2017). PPO became the default policy-gradient method, from simulated robots to the fine-tuning of language models on human feedback (Ouyang et al. 2022).

This section builds both. A two-action policy shows why parameter distance is the wrong quantity. A change of measure then lets a batch collected by the old policy evaluate a new one, exactly but with a variance that grows with the episode length, and keeping one ratio per step makes the reuse affordable. A short lemma says what that approximation leaves out; TRPO bounds the error and PPO clips it. On CartPole we then measure how many environment steps reuse saves, what goes wrong without the clip, and which numbers to watch while a policy trains.

```
import gymnasium as gym
```

```
torch.set_num_threads(1)
```

`torch.set_num_threads(1)` keeps PyTorch from spreading the small matrix products of these networks over many cores, which only slows them down.

15.2.1 Parameter Space versus Policy Space

Since we update the parameters of the policy, we should ask what a parameter update does to the policy itself. A family of policies with a single parameter θ and two actions answers it; the example comes from Joshua Achiam's lectures on policy optimization (Achiam 2017). Let

$$\pi_\theta(a) = \begin{cases} \sigma(\theta) & a = 1 \\ 1 - \sigma(\theta) & a = 2, \end{cases} \qquad \text{where } \sigma(\theta) = \frac{1}{1 + e^{-\theta}}. \tag{15.12}$$

Take two parameter updates of the same size, $\Delta\theta = 2$, from two different starting points:

$$\pi_{\theta=0}(a{=}1) = \sigma(0) = 0.50 \xrightarrow{\Delta\theta=2} \sigma(2) = 0.88, \qquad \pi_{\theta=6}(a{=}1) = \sigma(6) = 0.9975 \xrightarrow{\Delta\theta=2} \sigma(8) = 0. \tag{15.13}$$

The first update takes the agent from indifference between the two actions to a strong preference for one of them. The second changes nothing that an observer of the agent could detect. The slope of the map from parameter to policy says why: $\sigma'(\theta) = \sigma(\theta)(1 - \sigma(\theta))$ is 0.25 at $\theta = 0$ and about 0.0025 at $\theta = 6$. Figure 15.3 draws both updates.

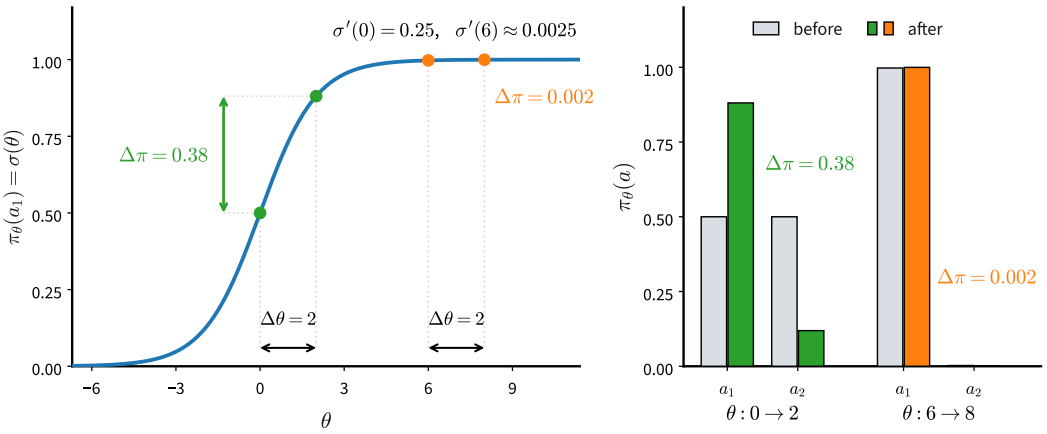


Figure 15.3: Two parameter updates of the same size, $\Delta\theta = 2$. Left: the map $\pi_\theta(a{=}1) = \sigma(\theta)$ with both updates and the slopes at their starting points. Right: the two action distributions before and after each update.

This difference makes on-policy training fragile. A large noisy update near $\theta = 0$ can push the policy into saturation. There the frequently chosen action has an almost-zero score, $\nabla_\theta \log \pi_\theta(a{=}1) = 1 - \sigma(\theta) \approx 0.0025$ at $\theta = 6$, while the other action has a score near -1 but is sampled once in four hundred steps. Later on-policy batches then carry too little evidence to reverse the update. A limit on the Euclidean length of the parameter step does not prevent this, because the same length is harmless at $\theta = 6$ and drastic at $\theta = 0$.

An update should therefore limit a distance between policies rather than between parameters. Within such a limit the optimizer may take a long parameter step where it changes the policy little, and, as we will see, it may take several steps on the same batch.

15.2.2 Importance Sampling

15.2.2.1 The Change of Measure

Every estimator since Section 14.6 has used trajectories sampled from the current policy, so a batch became useless after one gradient step. Suppose instead that the batch was collected by an earlier policy $\pi_{\theta_{\text{old}}}$. Can it still tell us how well an updated policy π_θ does?

Importance sampling gives an exact answer. For any function f of trajectories,

$$\mathbb{E}_{\tau\sim P(\cdot;\,\theta)}\big[f(\tau)\big] = \sum_{\tau} P(\tau;\theta)\, f(\tau) = \sum_{\tau} P(\tau;\theta_{\text{old}})\, \frac{P(\tau;\theta)}{P(\tau;\theta_{\text{old}})}\, f(\tau) = \mathbb{E}_{\tau\sim P(\cdot;\,\theta_{\text{old}})}\Big[\frac{P(\tau;\theta)}{P(\tau;\theta_{\text{old}})}\, f(\tau)\Big], \tag{15.14}$$

valid whenever every trajectory the new policy can produce has positive probability under the old one. Softmax policies satisfy this condition, because they never assign zero probability to an action. Applied with $f = R$, the return, the identity rewrites the objective as an expectation under the *old* policy:

$$J(\theta) = \mathbb{E}_{\tau\sim P(\cdot;\,\theta)}\big[R(\tau)\big] = \mathbb{E}_{\tau\sim P(\cdot;\,\theta_{\text{old}})}\Big[\frac{P(\tau;\theta)}{P(\tau;\theta_{\text{old}})}\, R(\tau)\Big]. \tag{15.15}$$

Thus data from the old policy, each trajectory weighted by its likelihood ratio, estimate the new policy's return without bias. The probability of a trajectory is a product of the start-state probability, the policy's action probabilities and the environment's transition probabilities, as in Section 14.6. In the ratio everything but the policy cancels, so the weight needs no model of the environment:

$$\frac{P(\tau;\theta)}{P(\tau;\theta_{\text{old}})} = \prod_{t=0}^{T-1} \frac{\pi_\theta(a_t \mid s_t)}{\pi_{\theta_{\text{old}}}(a_t \mid s_t)}. \tag{15.16}$$

15.2.2.2 The Exploding Product

The estimator is unbiased, but its variance grows with the length of the episode. Consider a policy that ignores the state, collects data with $\pi_{\theta_{\text{old}}} = (0.5, 0.5)$, and is evaluated at $\pi_\theta = (0.6, 0.4)$. Each factor of Equation 15.16 is 1.2 or 0.8 with equal probability, so its mean is 1 and its mean square is $(1.44 + 0.64)/2 = 1.04$. The factors of different steps are independent, so the trajectory weight W_T of an episode of T steps has mean 1 and variance

$$\text{Var}[W_T] = 1.04^T - 1,$$

which is about 0.5 for $T = 10$, about 50 for $T = 100$ and about 3×10^8 for CartPole's 500 steps. A policy that differs from the data's policy by ten percentage points per decision thus cannot be evaluated from whole CartPole episodes: a handful of trajectories with large weights decide the estimate.

15.2.2.3 The Per-Step Surrogate

The reward at step t depends only on the actions taken up to step t, so the exact correction needs only the ratios $\rho_0 \cdots \rho_t$ for that reward, where

$$\rho_t(\theta) = \frac{\pi_\theta(a_t \mid s_t)}{\pi_{\theta_{\text{old}}}(a_t \mid s_t)}. \tag{15.17}$$

The product $\rho_0 \cdots \rho_{t-1}$ of the earlier ratios corrects for the states: it turns the old policy's distribution of s_t into the new policy's. Dropping it, and keeping only ρ_t for the action at s_t,

gives a weight whose variance no longer grows with t. This one approximation evaluates the new policy's actions at the old policy's states. With an advantage estimate $\hat{A}_t^i$ for step t of trajectory i, the resulting *surrogate objective* averages over all $N = \sum_i T_i$ steps of a batch of n trajectories:

$$\hat{L}(\theta) = \frac{1}{N} \sum_{i=1}^{n} \sum_{t=0}^{T_i - 1} \rho_t^i(\theta)\, \hat{A}_t^i. \tag{15.18}$$

The average over steps rather than over trajectories is what the code computes, and it makes the divergence constraint below a mean over states. At $\theta = \theta_{\text{old}}$ every ratio equals one and $\nabla_\theta \rho_t = \nabla_\theta \log \pi_\theta(a_t \mid s_t)$, so the gradient of $\hat{L}$ there points in the same direction as the estimator of Section 14.7: each score weighted by its advantage, with the factor γ^t dropped, as every implementation since then has done. With advantages standardized to mean zero, $\hat{L}(\theta_{\text{old}}) = 0$ exactly. Away from θ_{old} the surrogate becomes less accurate, and the next question is how far it can be trusted.

15.2.3 Trust Regions

15.2.3.1 The Performance Difference Lemma

The surrogate evaluates the new policy's actions at the old policy's states. To see what this substitution changes, we compare the two returns exactly. Write $A^{\text{old}}(s,a) = Q^{\pi_{\theta_{\text{old}}}}(s,a) - V^{\pi_{\theta_{\text{old}}}}(s)$ for the old policy's advantage: how much better action a is than the old policy's average at s, when the old policy acts afterward. The following lemma holds for episodes that end at a terminal state, and for continuing tasks with $\gamma < 1$. Our CartPole runs bootstrap at the 500-step time limit, as in Section 14.8, so the objective they optimize is the continuing one.

Proposition (performance difference lemma, Kakade and Langford (2002)).

$$J(\theta) - J(\theta_{\text{old}}) = \mathbb{E}_{\tau \sim P(\cdot;\theta)} \Big[\sum_{t \ge 0} \gamma^t\, A^{\text{old}}(s_t, a_t) \Big]. \tag{15.19}$$

Proof. Write V for $V^{\pi_{\theta_{\text{old}}}}$. Along any trajectory the sum $\sum_t \gamma^t (r_t + \gamma V(s_{t+1}) - V(s_t))$ telescopes: the interior V terms cancel in pairs, and the last one is zero at a terminal state or vanishes as $t \to \infty$ when $\gamma < 1$. What remains is $R(\tau) - V(s_0)$. Now take expectations under $\tau \sim P(\cdot;\theta)$. Given (s_t, a_t), the expectation of $r_t + \gamma V(s_{t+1})$ over the next state is $Q^{\pi_{\theta_{\text{old}}}}(s_t, a_t)$, by the Bellman equation of the old policy, so the term at step t becomes $\gamma^t\, \mathbb{E}[A^{\text{old}}(s_t, a_t)]$. On the other side, $\mathbb{E}[R(\tau)] = J(\theta)$, and $\mathbb{E}[V(s_0)] = J(\theta_{\text{old}})$ because both policies draw s_0 from the same start distribution. ■

The lemma says that the new policy improves by the old policy's advantages, summed along the new policy's own trajectories. Grouping its terms by state makes the comparison with the surrogate exact. Let $d_\gamma^\pi(s) = (1-\gamma) \sum_{t \ge 0} \gamma^t \Pr(s_t = s)$ be the discounted state-occupancy distribution of Section 14.6, the distribution of states a policy visits with step t weighted by γ^t. Then

$$J(\theta) - J(\theta_{\text{old}}) = \frac{1}{1-\gamma} \sum_s d_\gamma^{\pi_\theta}(s) \sum_a \pi_\theta(a \mid s)\, A^{\text{old}}(s,a).$$

The surrogate computes this sum with the old policy's occupancy $d_\gamma^{\pi_{\theta_{\text{old}}}}$ in place of the new one's; that substitution is its only error. The error depends on how much the state distribution moves, and that depends on how much the policy changes, not on how far the parameters move.

15.2.3.2 The Monotonic-Improvement Bound

Replacing the occupancy in the last display defines the population version of the surrogate,

$$\bar{L}(\theta) = \frac{1}{1-\gamma} \sum_s d_\gamma^{\pi_{\theta_{\text{old}}}}(s) \sum_a \pi_\theta(a \mid s)\, A^{\text{old}}(s,a), \tag{15.20}$$

of which $\hat{L}$ is the sampled estimate, without the factor γ^t. Since the old policy's advantage averages to zero at every state, $\bar{L}(\theta_{\text{old}}) = 0$. How far may θ move before $\bar{L}$ stops predicting the improvement? The answer is a bound in terms of the largest change of the action distribution at any state.

Proposition (Kakade and Langford (2002); Schulman et al. (2015)). *For $\gamma < 1$ and $A_{\max} = \max_{s,a} |A^{old}(s,a)|$,*

$$J(\theta) \;\geq\; J(\theta_{\text{old}}) + \bar{L}(\theta) \;-\; \frac{4\gamma A_{\max}}{(1-\gamma)^2} \max_s D_{\text{KL}}\big(\pi_{\theta_{\text{old}}}(\cdot \mid s) \,\|\, \pi_\theta(\cdot \mid s)\big). \tag{15.21}$$

The surrogate is right up to a penalty proportional to the largest Kullback–Leibler divergence between the old and new action distributions at any state. Exercise 9 derives it by running the two policies side by side with coupled actions. Both sides of the bound equal $J(\theta_{\text{old}})$ at $\theta = \theta_{\text{old}}$, so any step that raises the right-hand side raises the true return: maximizing the bound gives monotonic improvement. The constant is too large to use as it stands. At $\gamma = 0.99$ it is $4 \cdot 0.99/0.01^2 \approx 40{,}000$ times $A_{\max}$, so the steps it certifies are tiny.

TRPO keeps the shape of the bound and replaces its pieces by quantities a batch can estimate. It maximizes the sampled surrogate subject to a budget Δ_{KL} on the mean divergence over the batch's states,

$$\max_\theta \; \hat{L}(\theta) \quad \text{subject to} \quad \frac{1}{N} \sum_{i,t} D_{\text{KL}}\big(\pi_{\theta_{\text{old}}}(\cdot \mid s_t^i) \,\|\, \pi_\theta(\cdot \mid s_t^i)\big) \leq \Delta_{\text{KL}}, \tag{15.22}$$

with a mean in place of the maximum over all states and a chosen budget, typically $\Delta_{\text{KL}} = 0.01$, in place of the conservative coefficient. It solves this problem with a second-order method, which exercise 10 builds from the pieces of this section. Figure 15.4 shows why the constraint lives in policy space.

15.2.3.3 The Clipped Objective

PPO replaces the constrained second-order problem by a first-order objective with a built-in limit. It does not clip the ratios themselves; it stops rewarding a sample once its ratio has moved beyond a band of half-width ϵ_{clip} around one:

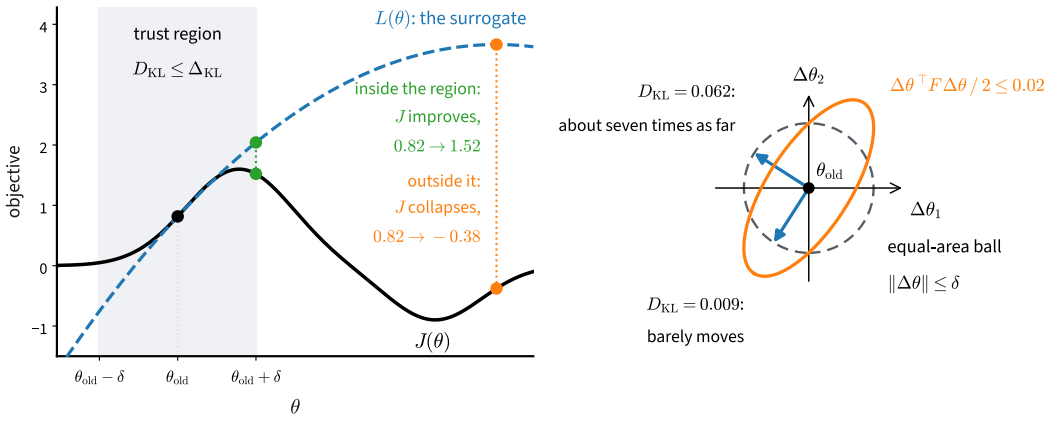


Figure 15.4: Bounding a policy update. Left: the surrogate L is tangent to the true objective J at θ_{old} but becomes inaccurate after a large policy change. The unconstrained maximizer of L lowers J from 0.82 to -0.38, whereas the best point inside the shaded trust region, $D_{\mathrm{KL}} \le \Delta_{\mathrm{KL}}$ (for one parameter an interval $\theta_{\mathrm{old}} \pm \delta$), raises it to 1.52. Right: for a three-action softmax policy, the second-order approximation $\frac{1}{2}\Delta\theta^\top F \Delta\theta \le 0.02$ of the divergence constraint, with F the Hessian of the divergence at θ_{old}, is an ellipse in parameter space. Two parameter steps of equal Euclidean length change the policy by divergences of 0.009 and 0.062: the same parameter distance moves the policy about seven times as far in one direction as in the other.

$$L^{\mathrm{CLIP}}(\theta) = \frac{1}{N} \sum_{i,t} \min \left(\rho_t^i(\theta)\, \hat{A}_t^i,\ \ \mathrm{clip}(\rho_t^i(\theta),\, 1 - \epsilon_{\mathrm{clip}},\, 1 + \epsilon_{\mathrm{clip}})\, \hat{A}_t^i \right). \tag{15.23}$$

Consider one sample. If $\hat{A}_t > 0$, its contribution grows with ρ_t only until $\rho_t = 1 + \epsilon_{\mathrm{clip}}$; beyond that the clipped term is the smaller one and its gradient is zero. If $\hat{A}_t < 0$, the threshold is $1 - \epsilon_{\mathrm{clip}}$. With the usual $\epsilon_{\mathrm{clip}} = 0.2$ and an action the old policy chose with probability 0.5, a sample with positive advantage stops rewarding the update once the new policy gives that action probability 0.6; with negative advantage, once the probability falls to 0.4. The minimum makes the limit one-sided: a change that lowers the objective is always counted, while further movement in the favorable direction adds nothing. Figure 15.5 plots one sample's contribution for both signs of the advantage.

Because the ratio itself is not clipped, it can still leave the band, for two reasons. All samples share the network's parameters, so a step taken for one sample moves the ratios of the others; and Adam's momentum keeps moving the parameters for a few steps after a sample's gradient has become zero. Engstrom et al. (2020) measured that PPO's ratios do leave the band and that its divergence per update is not held below any fixed budget. The clip is therefore an incentive rather than a constraint, and unlike Equation 15.21 it guarantees no improvement. What it does permit is several passes of optimization over one batch without rewarding ever larger changes of individual action probabilities.

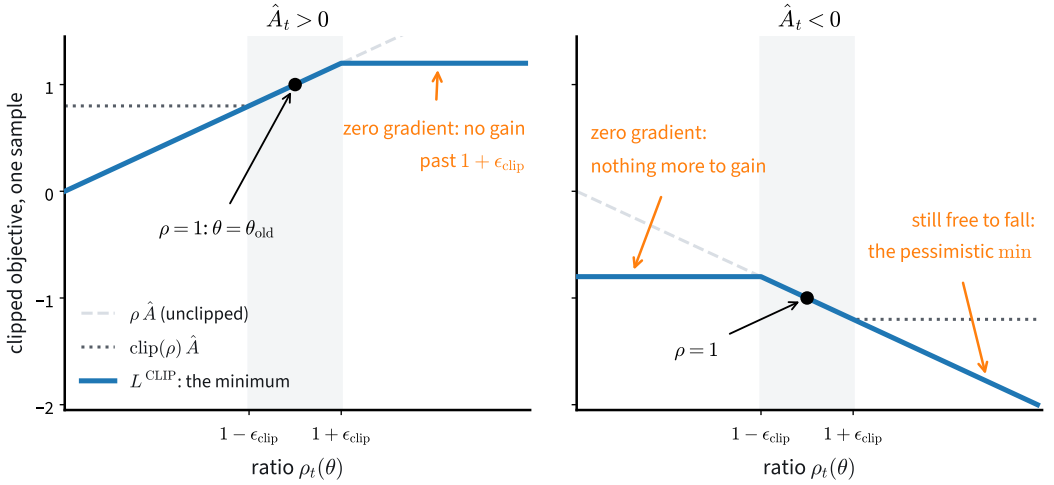


Figure 15.5: One sample's contribution to the clipped objective Equation 15.23 against the ratio ρ_t. For positive advantage it stops increasing above $1+\epsilon_{\text{clip}}$; for negative advantage it stops increasing below $1-\epsilon_{\text{clip}}$. On the other side of one the minimum keeps the unclipped term, so a change that lowers the objective is never hidden.

15.2.3.4 The Entropy Bonus

Implementations often add a small *entropy bonus*, a weight `entropy_coef` times the mean entropy of the action distributions at the batch's states. The bonus pushes probabilities away from zero and one, where, as in the two-action example, the frequently chosen action has a small score and recovery is slow. The entropy of a distribution over K actions is $\log K$ minus its divergence from the uniform distribution, so the bonus is a penalty on the distance from the uniform policy.

15.2.4 PPO in Practice

15.2.4.1 The Advantage Estimate

Any advantage estimate can fill the place of $\hat{A}$ in Equation 15.23. We use generalized advantage estimation (GAE) from Section 15.1: `Batch.gae` sums the TD errors $\delta_t = r_t + \gamma\hat{V}(s_{t+1}) - \hat{V}(s_t)$ of the critic backward through each episode with weights $(\gamma\lambda)^k$, and λ sets how far ahead each advantage looks. That section found the error of this estimate smallest near $\lambda = 0.9$ to 0.95; 0.95 is also the setting of the PPO paper. The advantages are computed with the critic as it stands before it is fit to the batch, the order of the GAE paper (Schulman et al. 2016). `train_ac` of Section 15.1 used the opposite order, because its actor's weight was the one-step TD error, which rests entirely on the critic. At $\lambda = 0.95$ the advantage is mostly the batch's own sampled rewards and leans little on the critic, so the order changes no result of this section, and we follow the paper.

15.2.4.2 The Implementation

We keep the CartPole setup of Section 15.1. The agent is `ActorCritic.mlp(4, 2)` of Section 14.8: a policy network and a value network, each with one hidden layer of 64 tanh units and its

own Adam optimizer with learning rate 10^{-2}. Each update collects a batch of eight episodes, and the critic takes `critic_steps = 20` regression passes on it. Three hyperparameters are new: `num_epochs = 20` passes of the actor over each batch, the half-width `epsilon_clip = 0.2` of the band, and the entropy weight `entropy_coef = 0.01`. Twenty passes is also the CartPole setting of the RL Baselines3 Zoo (Raffin et al. 2021), but the zoo's learning rate is 10^{-3}, ten times smaller than ours. At our rate each pass moves the policy far, which makes training fast and is the setting in which the clip matters; the ablation below shows what happens without it.

```
gamma, lam, num_updates, batch_episodes = 0.99, 0.95, 60, 8
num_seeds, num_epochs, critic_steps = 8, 20, 20
epsilon_clip, entropy_coef = 0.2, 0.01
def cartpole_agent(seed):
    return d2l.ActorCritic.mlp(4, 2, seed=seed)
```

The per-sample objective is passed to the update as a function. `clipped_surrogate` is Equation 15.23, and `plain_surrogate`, the term $\rho_t \widehat{A}_t$ alone, serves the ablation below; both take the half-width ϵ_{clip} of the band, which only the first one uses.

```
def clipped_surrogate(rho, adv, epsilon):
    """PPO's per-sample objective eq_ppo_clip: min(rho A, clip(rho) A)."""
    return torch.minimum(rho * adv,
                         rho.clamp(1 - epsilon, 1 + epsilon) * adv)

def plain_surrogate(rho, adv, epsilon):
    """The unclipped per-sample objective rho A."""
    return rho * adv
```

`ppo_epochs` receives a batch's states and actions together with the two quantities frozen before reuse: the advantages and the collecting policy's log-probabilities `logp_old`. It takes `num_epochs` steps of the actor's optimizer `opt_pi` on the negated mean surrogate minus the entropy bonus. The entropy needs the whole distribution at each state, so the update computes the log-softmax over all actions once and reads $\log \pi_\theta(a_t \mid s_t)$ from it. Before each pass it records one row of diagnostics: the fraction of ratios outside the band, the sample mean of $\log \pi_{\theta_{\text{old}}} - \log \pi_\theta$, and the mean entropy. The second number is called the *approximate KL*: its expectation under the old policy is the divergence $D_{\text{KL}}(\pi_{\theta_{\text{old}}} \| \pi_\theta)$ at the visited states, although on a finite batch it can come out negative. Since the row is recorded before the pass, the first row is always zero, zero and the old policy's entropy.

```
def ppo_epochs(ac, obs, act, adv, logp_old, epsilon, num_epochs,
               entropy_coef=0.01, surrogate=clipped_surrogate):
    """num_epochs passes on the mean surrogate plus an entropy bonus, on one
    frozen batch of states and actions; returns [num_epochs, 3] numpy
    diagnostics, recorded before each pass: fraction of ratios outside the
    band, approximate KL, mean policy entropy."""
    obs, act, adv, logp_old = (torch.as_tensor(x)
                               for x in (obs, act, adv, logp_old))
    diag = []
    for _ in range(num_epochs):
```

```
        logp_all = torch.log_softmax(ac.policy(obs), dim=-1)
        logp = logp_all.gather(-1, act[:, None]).squeeze(-1)
        rho = torch.exp(logp - logp_old)
        entropy = -(logp_all.exp() * logp_all).sum(-1).mean()
        loss = -surrogate(rho, adv, epsilon).mean() - entropy_coef * entropy
        ac.opt_pi.zero_grad()
        loss.backward()
        ac.opt_pi.step()
        diag.append(torch.stack([((rho - 1).abs() > epsilon).float().mean(),
                                 (logp_old - logp).mean(), entropy]).detach())
    return torch.stack(diag).numpy()
```

The training loop is the actor-critic loop of Section 15.1 with its single actor step replaced by `ppo_epochs` and the advantages computed before the critic's passes. `rollout` plays eight episodes into a `Batch`. The loop then computes the advantages with `Batch.gae` and the current critic, and standardizes them over the batch with `standardize`. It also reads `logp_old` with `log_prob_np`, which returns $\log \pi(a \mid s)$ for given states and actions as a NumPy array. Next the critic takes its regression passes toward the λ-return target with `Batch.lambda_return` of Section 15.1 and `fit_value` of Section 14.8, and finally the actor takes its passes. Each update yields the environment steps spent so far and the batch's mean return. The diagnostics of each batch go to a list `diag` that the caller passes in; `CLIPPED`, `KL` and `ENTROPY` name their three columns.

```
CLIPPED, KL, ENTROPY = 0, 1, 2   # the columns of ppo_epochs' diagnostics

def train_ppo(seed, ac, diag, surrogate=clipped_surrogate,
              num_epochs=num_epochs):
    """PPO on CartPole: freeze the advantages and the collecting policy's
    log-probs, then take num_epochs surrogate passes on each batch. Appends
    each batch's diagnostics to diag; yields (env steps, mean return)."""
    rng, env = np.random.default_rng(seed), gym.make('CartPole-v1')
    env.reset(seed=seed)
    steps = 0
    for _ in range(num_updates):
        b = d2l.rollout(env, ac.act, batch_episodes, rng)
        adv = d2l.standardize(b.gae(ac.value_np, gamma, lam))  # pre-fit critic
        logp_old = ac.log_prob_np(b.obs, b.act)
        for _ in range(critic_steps):   # fresh lambda-return target, per pass
            d2l.fit_value(ac, b.obs, b.lambda_return(ac.value_np, gamma, lam))
        diag.append(ppo_epochs(ac, b.obs, b.act, adv, logp_old, epsilon_clip,
                               num_epochs, entropy_coef, surrogate))
        steps += len(b)
        yield steps, float(b.episode_returns().mean())
```

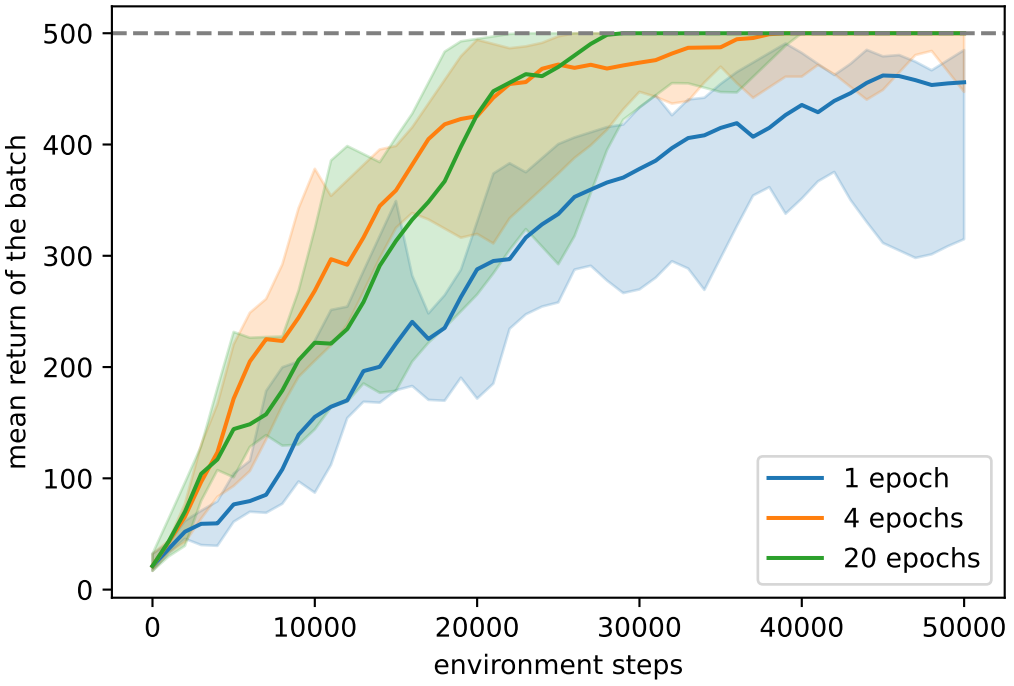
15.2.4.3 Reuse against Environment Steps

How many environment steps does reuse save? We train clipped PPO with one, four and twenty passes per batch, on eight seeds each, stop each run once it has spent 50,000 environment steps, and plot the batch's mean return against the steps spent so far. With one pass every ratio equals one and the clip never acts, so that arm is close to the λ sweep of Section 15.1, with an entropy bonus, no cap on the gradient, and the advantages computed

before the critic's passes. As there, `d2l.on_grid` places each run on a common grid of steps. These runs need no diagnostics, so each gets an empty list.

```
budget = 50_000
grid = np.arange(0, budget + 1, 1000)
reuse = {}
for e in (1, 4, 20):
    runs_e = []
    for s in range(num_seeds):
        run = []
        for row in train_ppo(s, cartpole_agent(s), [], num_epochs=e):
            run.append(row)                    # (env steps, mean return)
            if row[0] >= budget:
                break
        runs_e.append(np.array(run))
    reuse[f'{e} epoch' + 's' * (e > 1)] = d2l.on_grid(runs_e, grid, 1)
```

```
d2l.plot_curves(reuse, xlabel='environment steps',
                ylabel='mean return of the batch', x=grid, reference=500)
```



Reusing each batch for four epochs reaches the ceiling in about half the environment steps that one epoch needs, roughly 25,000 against 50,000 or more, and in about half the updates; within the 50,000-step budget the one-epoch median stays just short of the ceiling. Going on to twenty epochs changes little by comparison, and no clipped run collapses. The gain comes from the batch: a batch of a few thousand steps holds more information than one gradient step can extract, and the clip lets the optimizer take several steps without moving any action probability too far.

15.2.4.4 The Clipping Ablation

Four passes are enough to collect most of the gain; at our learning rate, twenty passes move the policy a long way within one batch. Does the clip matter there? We train twenty-epoch PPO on the same eight seeds with the clip on, keeping the trained agents for later checks and the full diagnostics for the next subsection:

```
arms = {'clipped (PPO)': clipped_surrogate, 'no clip': plain_surrogate}
agents = {arm: [cartpole_agent(s) for s in range(num_seeds)] for arm in arms}
runs, diag = {}, {arm: [] for arm in arms}
runs['clipped (PPO)'] = np.array(
    [list(train_ppo(s, agents['clipped (PPO)'][s], diag['clipped (PPO)']))
     for s in range(num_seeds)])
```

Then the control, identical in every line except that the surrogate is unclipped. Each arm's diagnostics are then reshaped to (seed, update, epoch, column):

```
runs['no clip'] = np.array(
    [list(train_ppo(s, agents['no clip'][s], diag['no clip'],
                    surrogate=plain_surrogate))
     for s in range(num_seeds)])
diag = {arm: np.reshape(d, (num_seeds, num_updates, num_epochs, 3))
        for arm, d in diag.items()}
```

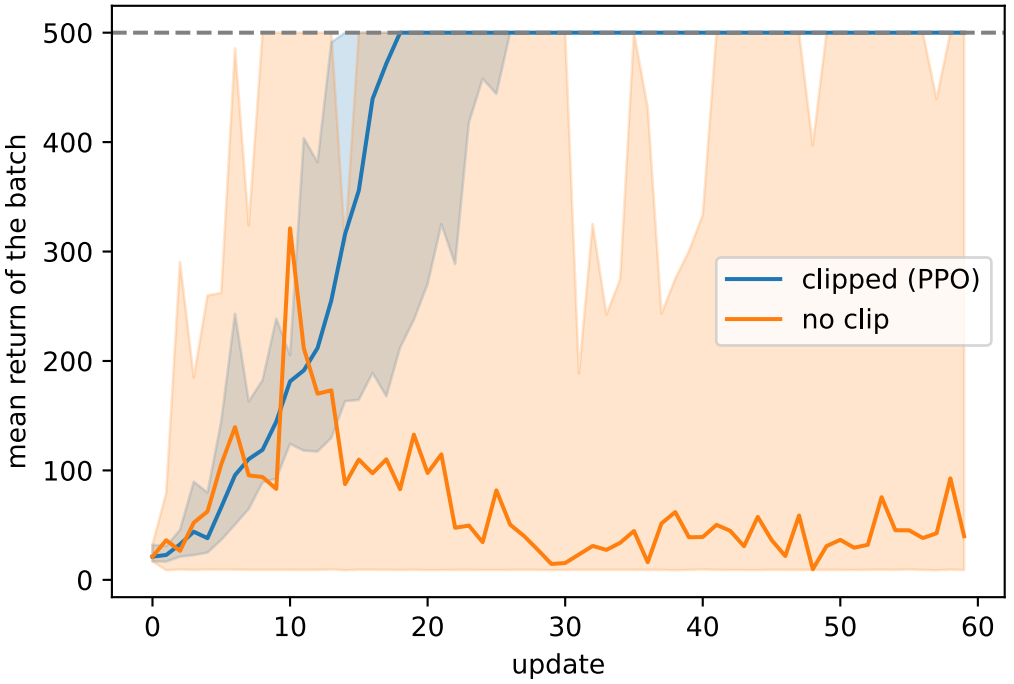
Two criteria summarize each run. A run *holds* the task if its batch return, once it first reaches 475, never again falls below 250, half the maximum. A run ends *dead* if its last ten updates average below a return of 100:

```
def holds(ret):   # reaches 475, then never falls below 250 again
    hit = np.flatnonzero(ret >= 475)
    return len(hit) > 0 and ret[hit[0]:].min() >= 250

for name, r in runs.items():
    held = sum(holds(ret) for ret in r[:, :, 1])
    dead = r[:, -10:, 1].mean(axis=1) < 100
    print(f'{name:>13}: {held} of {num_seeds} seeds hold the task, '
          f'{int(dead.sum())} end dead')
```

```
clipped (PPO): 8 of 8 seeds hold the task, 0 end dead
      no clip: 0 of 8 seeds hold the task, 4 end dead
```

```
d2l.plot_curves({name: r[:, :, 1] for name, r in runs.items()},
                xlabel='update', ylabel='mean return of the batch',
                reference=500)
```



At twenty epochs and a learning rate of 10^{-2}, every clipped seed reaches the ceiling and holds it. Without the clip, over 64 seeds outside the notebook in each framework, about one run in twenty holds the task and about half end dead, most of them at a return of about 9, the length of an episode in which the cart is pushed the same way at every step. Repeated unclipped passes drive the policy into saturation, where, as in the two-action example, the gradient of the chosen action's log-probability is small; the next batches are then collected by the saturated policy, and they rarely contain the actions that would reverse it. The PPO paper's own ablation found the same: its variant with neither a clip nor a penalty was the worst of all those it tested (Schulman et al. 2017). The large learning rate is what makes the clip necessary. At an actor learning rate of 10^{-3}, ten times smaller, unclipped PPO no longer collapses on any of ten seeds and reaches the ceiling as fast as clipped PPO, and at 3×10^{-4} the clip almost never acts. Both rates are slower than ours, however: they need about 35,000 to 40,000 and about 85,000 to 90,000 steps to the ceiling, against about 24,000 here. The large rate makes training fast, and the clip makes the large rate safe.

How often does the clip act? Each update checks every sample once per epoch, and the first epoch begins with $\rho_t = 1$ for every sample. We report the fraction of all per-epoch checks outside the band and the same fraction at the last epoch of each batch:

```
for name, d in diag.items():
    print(f'{name:>13}: ratio checks outside the band: '
          f'{d[..., CLIPPED].mean():.1%} across all epochs')
    print(f'{"":>13}  {d[:, :, -1, CLIPPED].mean():.1%} at the last epoch '
          f'of each batch')
```

```
clipped (PPO): ratio checks outside the band: 5.3% across all epochs
                5.0% at the last epoch of each batch
      no clip: ratio checks outside the band: 17.6% across all epochs
```

```
25.1% at the last epoch of each batch
```

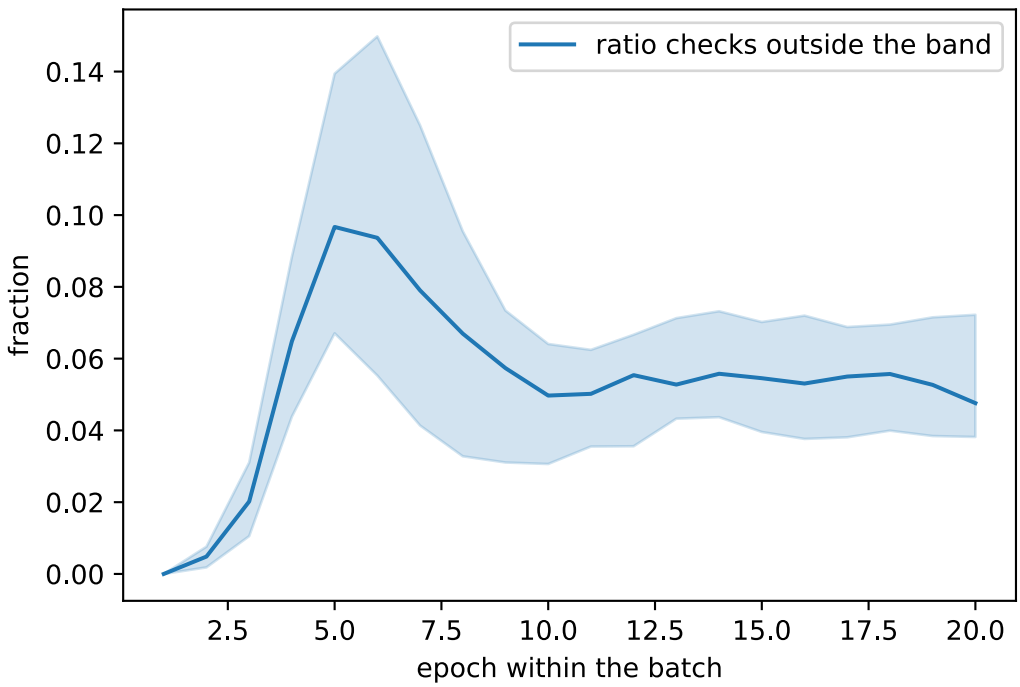
For the clipped runs both counts are near one check in twenty. In the unclipped control ratios leave the band several times as often, and more often at the last epoch of a batch than on average, because nothing stops them from moving further at every pass.

15.2.4.5 Training Diagnostics

Section 14.8 showed that the value of a policy-gradient loss says nothing about progress: it stays near zero while the return climbs from 20 to 500. What deserves watching are measurements of the update itself, and `diag` already holds them for every run, seed, update and epoch. We first average over the updates and look at one batch's twenty epochs:

```
clipped = diag['clipped (PPO)']
d2l.plot_curves({'ratio checks outside the band':
                 clipped[..., CLIPPED].mean(1)},
                xlabel='epoch within the batch', ylabel='fraction',
                x=np.arange(1, num_epochs + 1))
kl = np.median(clipped[..., KL].mean(1), axis=0)
print(f'approximate KL: peak {kl.max():.4f} at epoch {kl.argmax() + 1}, '
      f'{kl[-1]:.4f} at epoch {num_epochs}')
for name, d in diag.items():
    print(f'{name:>13}: approximate KL at the last epoch, median '
          f'{np.median(d[:, :, -1, KL]):.3f}')
```

```
approximate KL: peak 0.0085 at epoch 6, 0.0053 at epoch 20
clipped (PPO): approximate KL at the last epoch, median 0.003
      no clip: approximate KL at the last epoch, median 0.139
```

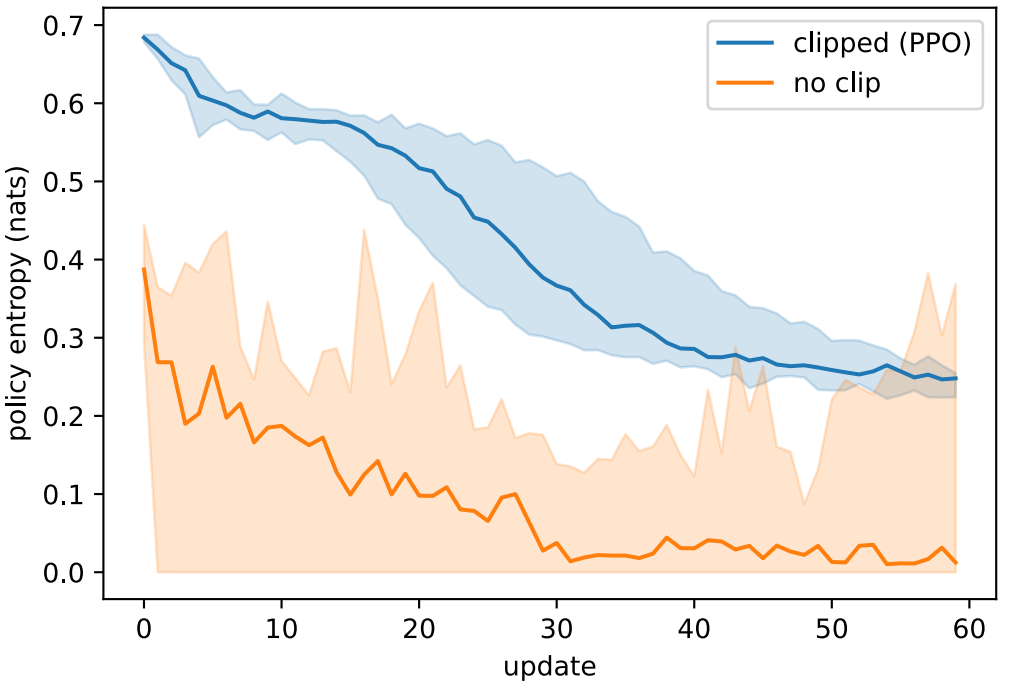


Within a batch the fraction outside the band is exactly zero at the first epoch, where every ratio equals one. It rises to a peak around the fifth or sixth epoch and then falls to about half the peak by the twentieth. The fall comes from the one-sided minimum. A sample whose ratio has left the band on the losing side keeps the gradient of its unclipped term, which pulls the ratio back, while a sample past the band on the winning side gets no gradient and stays where it is. The approximate KL has the same shape and is about ten times smaller; the cell prints its peak. Without the clip the divergence keeps growing at every pass, and at the last epoch of a batch it is typically tens of times the clipped run's.

Across updates, the entropy of the clipped policies tells how decisive they have become:

```
d2l.plot_curves({name: d[..., ENTROPY].mean(-1) for name, d in diag.items()},
                xlabel='update', ylabel='policy entropy (nats)')
print(f'clipped: entropy {clipped[:, :5, :, ENTROPY].mean():.2f} over the '
      f'first five updates, {clipped[:, -5:, :, ENTROPY].mean():.2f} over the '
      f'last five')
```

```
clipped: entropy 0.65 over the first five updates, 0.25 over the last five
```



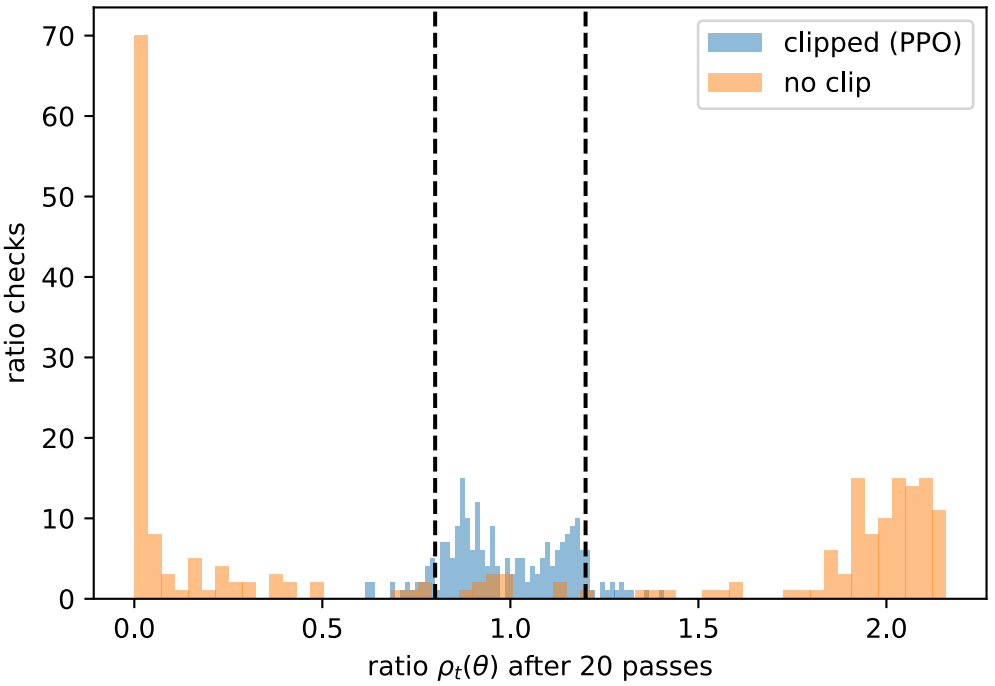
The entropy of the clipped runs falls from about 0.65 nats to about 0.25: the policy becomes more decisive as it learns, and the entropy bonus slows this without stopping it. The unclipped runs lose their entropy much faster, which is the saturation that ended them.

The summary fractions hide how the ratios are distributed. We take two identical fresh agents at the start of training, where the drift within a batch is largest, give both the same batch with the same frozen advantages and log-probabilities, and take twenty passes with the clip on in one and off in the other, reading every ratio after each pass. A third twin, `collector`, collects the batch and stays as it is, the old policy of every ratio:

```
probe = {arm: cartpole_agent(2) for arm in arms}   # twins
collector = cartpole_agent(2)                      # a third twin, kept fresh
env = gym.make('CartPole-v1')
env.reset(seed=2)
batch = d2l.rollout(env, collector.act, batch_episodes,
                    np.random.default_rng(2))
adv = d2l.standardize(batch.gae(collector.value_np, gamma, lam))
logp_old = collector.log_prob_np(batch.obs, batch.act)
rhos = {}
for arm, surrogate in arms.items():
    R = []
    for _ in range(num_epochs):   # one pass at a time, ratios read after
        ppo_epochs(probe[arm], batch.obs, batch.act, adv, logp_old,
                   epsilon_clip, 1, entropy_coef, surrogate)
        R.append(np.exp(probe[arm].log_prob_np(batch.obs, batch.act)
                        - logp_old))
    rhos[arm] = np.array(R)
```

`d2l.plot_hist` (the online appendix "Utility Functions and Classes") overlays the histograms of the two sets of ratios after the last pass and marks the band edges with dashed lines:

```
d2l.plot_hist({arm: r[-1] for arm, r in rhos.items()},
              xlabel=rf'ratio $\rho_t(\theta)$ after {num_epochs} passes',
              ylabel='ratio checks',
              vlines=(1 - epsilon_clip, 1 + epsilon_clip))
```

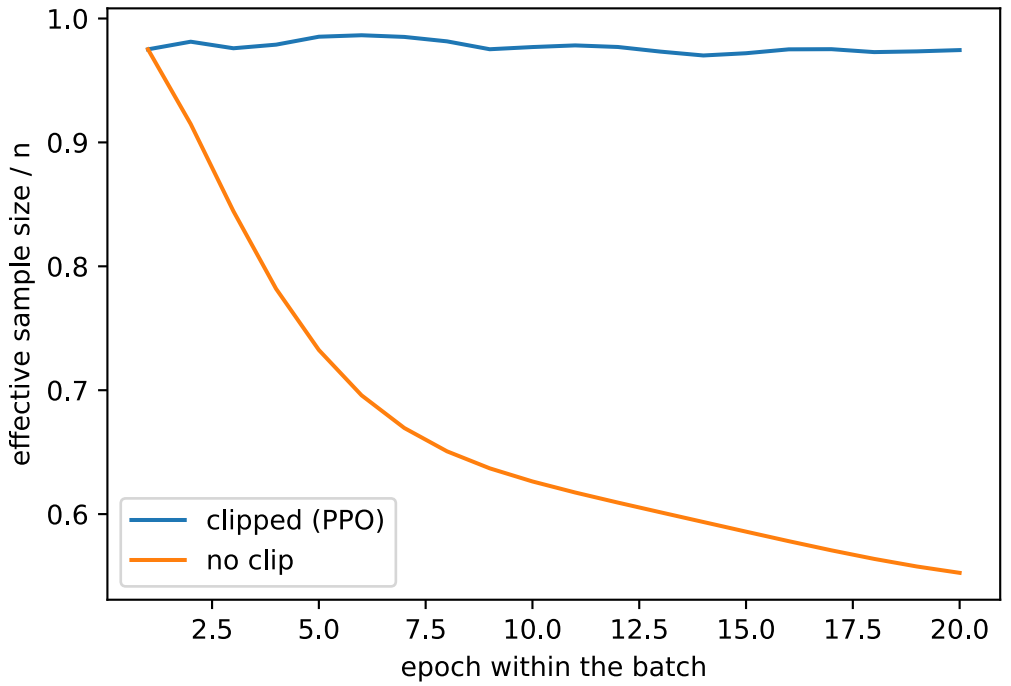


After twenty passes the clipped agent's ratios stay near one, with a minority beyond the

dashed band edges. The unclipped ratios pile up near zero, and most of the others lie far above the band, up to about two or three. A ratio of two or three is $1/\pi_{\theta_{\text{old}}}(a)$ for an action that the fresh policy chose with probability a half or a third and the new policy chooses almost always; a ratio near zero belongs to the other action. How much is such a reweighted batch still worth? The *effective sample size* of n importance weights answers this, $N_{\text{eff}} = 1/\sum_j \bar{w}_j^2$, where $\bar{w}_j$ is the ratio of step j divided by the sum of all the batch's ratios. It equals n when all weights are equal and one when a single weight dominates.

```
ess = {}
for arm, r in rhos.items():
    w = r / r.sum(axis=1, keepdims=True)   # normalized weights per epoch
    ess[arm] = 1 / (w ** 2).sum(axis=1) / r.shape[1]
d2l.plot_curves(ess, xlabel='epoch within the batch',
                ylabel='effective sample size / n',
                x=np.arange(1, num_epochs + 1))
print(f'after {num_epochs} epochs the batch is worth '
      + ' vs '.join(f'{v[-1]:.0%} ({k})' for k, v in ess.items())
      + f' of its {len(batch)} steps')
```

```
after 20 epochs the batch is worth 97% (clipped (PPO)) vs 55% (no clip) of its 226 steps
```



With the clip the weights stay nearly equal through the twentieth epoch. Without it the effective sample size falls well below the clipped arm's, to about half the batch. Saturation explains the value. Suppose the new policy picks an action $a^*(s)$ with certainty. A sample at s then has weight $1/\pi_{\theta_{\text{old}}}(a^* \mid s)$ if its action was a^*, which happens with probability $\pi_{\theta_{\text{old}}}(a^* \mid s)$, and weight zero otherwise. The mean weight is one and the mean squared weight is $1/\pi_{\theta_{\text{old}}}(a^* \mid s)$, so as the batch grows N_{eff}/n approaches the harmonic mean of

$\pi_{\theta_{\text{old}}}(a^* \mid s)$ over the visited states. For a fresh policy that is nearly uniform over two actions, that is about one half. To check this we need the probabilities of all actions, not only of the sampled one; `action_probs` returns them as a NumPy array, and the untrained twin `collector` is the collecting policy.

```
def action_probs(ac, obs):
    """pi(.|s) for a numpy batch of states, as numpy."""
    with torch.inference_mode():
        return torch.softmax(ac.policy(torch.as_tensor(obs)), -1).numpy()
```

```
p_old = action_probs(collector, batch.obs)     # the collecting policy
p_new = action_probs(probe['no clip'], batch.obs)
a_star = p_new.argmax(axis=1)                  # the unclipped choice
p_star = p_old[np.arange(len(batch)), a_star]
print(f'unclipped policy: median probability of its choice '
      f'{np.median(p_new.max(axis=1)):.3f}')
print(f'harmonic mean of the old probability of that choice: '
      f'{1 / (1 / p_star).mean():.2f}')
```

```
unclipped policy: median probability of its choice 0.996
harmonic mean of the old probability of that choice: 0.49
```

The unclipped policy is close to deterministic, and the harmonic mean lies within about a tenth of the unclipped arm's effective sample size, slightly below it because the policy is not quite deterministic. The effective sample size counts how evenly the weights are spread; it ignores the advantages and the states the new policy would visit, so it complements the divergence and the clip fraction rather than replacing them.

Finally, a trained clipped agent, acting greedily on the most probable action to remove the noise of sampling, should balance the pole to the time limit:

```
ac, env = agents['clipped (PPO)'][0], gym.make('CartPole-v1')
env.reset(seed=2)
score = d2l.evaluate(env, ac.act_greedy, num_episodes=100)
print(f'greedy mean return over 100 episodes: {score:.0f}')
```

```
greedy mean return over 100 episodes: 500
```

15.2.4.6 Vectorized Collection and Minibatches

Our loop plays whole episodes in one environment and takes full-batch passes. Production implementations such as `cleanrl/ppo.py` (Huang, Rousslan F. J. Dossa, et al. 2022) step N copies of the environment in parallel for a fixed horizon T, producing an $N \times T$ rectangle of transitions, and cut every episode that is still running at the rectangle's edge. At the cut the critic supplies the value of the unrecorded future, the same bootstrap that Section 14.8 used at CartPole's time limit. We collect one such rectangle with the trained agent, eight environments for 32 steps:

```
envs = gym.vector.SyncVectorEnv([lambda: gym.make('CartPole-v1')] * 8)
obs, _ = envs.reset(seed=0)
rng, steps = np.random.default_rng(3), []
for _ in range(32):                       # the fixed 32 x 8 rectangle
    u = rng.random((len(obs), 1))         # inverse-CDF sampling, per env
    act = (u > action_probs(ac, obs).cumsum(1)[:, :-1]).sum(1)
    obs, rew, term, trunc, _ = envs.step(act)
    steps.append((rew, term | trunc))
rew, done = map(np.array, zip(*steps))
print(f'a fixed {rew.shape} rectangle of steps, '
      f'{int(done.sum())} episode boundaries inside it')
print(f'V at the cut, the value of the unrecorded future: '
      f'{ac.value_np(obs).round(0)}')
```

```
a fixed (32, 8) rectangle of steps, 0 episode boundaries inside it
V at the cut, the value of the unrecorded future: [100. 100. 100. 100. 100. 100. 100. 100.]
```

The trained agent does not fall within 32 steps, so all eight trajectories end at the cut, and their values are close to $1/(1-\gamma) = 100$, the discounted value of balancing forever. Without the bootstrap, GAE would treat each cut as a fall and give the last actions before it large negative advantages. Gymnasium's vector environments also restart a finished episode by themselves: the step after an episode ends returns the first observation of a new episode with reward zero and ignores the action, so implementations mask that step out of the advantages and the loss. Finally, production code shuffles the rectangle into minibatches and takes a few epochs over them, typically four, so each ratio is checked four times rather than twenty; the smaller steps need less memory and add gradient noise. Exercise 8 builds this loop.

PPO lets an agent take several steps on each batch, and on CartPole four epochs reach the ceiling in half the environment steps of one. Its clip limits how far each step moves the action probabilities, but it bounds the path, not the destination. The entropy of the clipped runs still falls from 0.65 to 0.25 nats, because the objective is still the return with a small entropy bonus, whose optimum is nearly deterministic. Section 15.3 changes the objective instead: it keeps the policy near a fixed reference, which moves the optimum and not only the path.

15.2.5 Summary

An on-policy batch of thousands of environment steps should inform more than one gradient step, but equal steps in parameter space can change a policy by very different amounts. Importance sampling evaluates a new policy on old data exactly, with a variance that grows exponentially with the episode length; keeping one ratio per step removes that growth and evaluates the new policy's actions at the old policy's states instead. The performance difference lemma shows that this substitution is the surrogate's only error, and TRPO bounds it by constraining the divergence between old and new action distributions. PPO instead clips each sample's contribution to the objective once its ratio leaves a band around one, which lets a few epochs reuse each batch: on CartPole four epochs reach the ceiling in about half the environment steps of one, and the clip keeps a large learning rate from collapsing the policy. The clip fraction, the approximate KL, the entropy and the effective sample size of the ratios show how far each update moved the policy.

15.2.6 Exercises

1. **Step limits for one logit.** ⊠ Take the two-action policy $\pi_\theta(a{=}1) = \sigma(\theta)$ of Figure 15.3 and three ways of limiting an update $\theta \to \theta + \Delta$: a trust region on the divergence, PPO's clip, and a cap on $|\Delta|$.
   1. Expand $D_{\mathrm{KL}}(\pi_\theta \| \pi_{\theta+\Delta})$ to second order in Δ and show that it equals $\frac{1}{2}\sigma'(\theta)\,\Delta^2$. Compute the exact divergence of the two updates with $\Delta = 2$ drawn in the figure, from $\theta = 0$ and from $\theta = 6$. Which of them stays inside a trust region with $\Delta_{\mathrm{KL}} = 0.02$?
   2. Use the second-order form to find the largest $|\Delta|$ that this trust region permits at $\theta = 0$ and at $\theta = 6$. At $\theta = 6$, compute the exact divergence of the two steps of that size, one in each direction. Why is the second-order form much less accurate there than at $\theta = 0$?
   3. Now let the clip of Equation 15.23 with $\epsilon_{\text{clip}} = 0.2$ limit the step, for one sample with positive advantage. The update raises the probability of the sampled action, so $\Delta > 0$ when the sample is $a = 1$ and $\Delta < 0$ when it is $a = 2$. Find the largest $|\Delta|$ over which the clipped objective still increases, for both actions at $\theta = 0$ and at $\theta = 6$.
   4. Which of the three limits permits large parameter steps where the policy is insensitive to θ and only small ones where it is sensitive? At $\theta = 6$, for which sampled action is the clip looser than the trust region, and for which is it tighter?

2. **Exploding product of ratios.** A policy that ignores the state collects episodes with $\pi_{\theta_{\text{old}}}$ and the data are used to evaluate π_θ. Write W_T for the trajectory ratio Equation 15.16 of an episode of T steps.
   1. Show that $\mathbb{E}[W_T] = 1$ and $\mathrm{Var}[W_T] = (1+\chi^2)^T - 1$ under the old policy, where $\chi^2 = \sum_a \big(\pi_\theta(a) - \pi_{\theta_{\text{old}}}(a)\big)^2 / \pi_{\theta_{\text{old}}}(a)$. Check the value $1.04^T - 1$ of the text for $\pi_{\theta_{\text{old}}} = (0.5, 0.5)$ and $\pi_\theta = (0.6, 0.4)$.
   2. As the number n of episodes grows, the effective sample size of the weights W_T approaches $n/\mathbb{E}[W_T^2]$. For the two policies of (a), for which n does this limit equal one at $T = 200$? Simulate one batch of that many episodes (the number of steps with $a = 1$ in an episode is binomial) and compute its effective sample size. Why does the batch report a much larger effective sample size than the limit? Which of the two describes the variance of the importance-weighted estimate?
   3. The per-step surrogate Equation 15.18 weights each step by a single ratio ρ_t. What is the variance of that weight here, and which approximation in Equation 15.18 makes it independent of T?

3. code **Entropy of the dead seeds.** The ablation of Section 15.2.4.4 explains the unclipped collapses by saturation. The array `diag['no clip'][..., ENTROPY].mean(-1)` holds the mean policy entropy over the epochs of each batch. Without new training, split the unclipped seeds into dead and surviving ones by the criterion of the ablation, a mean return below 100 over the last ten updates, and plot the entropy of both groups against the update. Do the dead seeds end with a saturated policy?

4. **What the diagnostics estimate.** ⊠ Fix a state s, draw $a \sim \pi_{\theta_{\text{old}}}(\cdot \mid s)$, and let $\rho = \pi_\theta(a \mid s)/\pi_{\theta_{\text{old}}}(a \mid s)$.
   1. The approximate KL of `ppo_epochs` averages $k_1 = -\log \rho$ over the batch. Show that $\mathbb{E}[k_1] = D_{\mathrm{KL}}(\pi_{\theta_{\text{old}}}(\cdot \mid s) \| \pi_\theta(\cdot \mid s))$. Then show that $k_3 = (\rho - 1) -$

$\log \rho$ has the same expectation and is nonnegative for every sample. Expand both estimators to second order in $\rho - 1$. Why does k_3 have a much smaller variance than k_1 when the policy changes little? `cleanrl/ppo.py` logs both.

2. Problem 2b uses the limit $n/\mathbb{E}[w^2]$ of the effective sample size of n independent weights w with $\mathbb{E}[w] = 1$. Derive this limit with the law of large numbers. Then show that $\mathbb{E}[\rho^2] = 1+\chi^2$, so that the effective sample size of the weights ρ divided by n approaches $1/(1 + \chi^2)$, where

$$\chi^2 = \sum_a \frac{\left(\pi_\theta(a \mid s) - \pi_{\theta_{\text{old}}}(a \mid s)\right)^2}{\pi_{\theta_{\text{old}}}(a \mid s)}.$$

Show that $\chi^2 \approx 2D_{\text{KL}}$ to second order in the policy change.

3. With states drawn from a batch, χ^2 becomes its average over the visited states. Which values of χ^2 do the two fractions `ess[name][-1]` of Section 15.2.4.5 imply? Which approximate KL does the second-order relation of (b) predict for each? Compare each prediction with the k_1 estimate of the same ratios, `(-np.log(rhos[arm][-1])).mean()` for each `arm`. For which agent does the second-order relation fail? Besides the size of the policy change, which assumption of (b) do ratios violate that are measured on the very batch the policy was trained on?

Adapted from John Schulman, Approximating KL Divergence, estimators k_1 and k_3.

5. **Bootstrapping at the cut.** In Section 15.2.4.6, all eight trajectories of the 32×8 rectangle end at the collection horizon T, where $\hat{V}(s_T) \approx 100$. Suppose that GAE with $\gamma = 0.99$ and $\lambda = 0.95$ treated each cut as a termination and used $\hat{V}(s_T) = 0$ there.

   1. The GAE advantage of a step before the cut is $\hat{A}_t = \sum_{l=0}^{T-1-t} (\gamma\lambda)^l \delta_{t+l}$ with $\delta_t = r_t + \gamma\hat{V}(s_{t+1}) - \hat{V}(s_t)$. Show that $\hat{V}(s_T)$ enters only through δ_{T-1}, and that the advantage $\hat{A}_{T-1-k}$ of the step k steps before the cut therefore shifts by $-(\gamma\lambda)^k\gamma\hat{V}(s_T)$. Evaluate the shift for $k \in \{0, 10, 31\}$ with $\hat{V}(s_T) = 100$.
   2. For which k does the shift exceed 1, the reward of one step? Which fraction of the steps of a trajectory is shifted by more than that with the horizon $T = 32$ used there, and with $T = 128$, the default of `cleanrl/ppo.py`?
   3. The CartPole observation does not contain the time since the rectangle began. What does a policy update with the shifted advantages teach the policy about the actions it took shortly before each cut?

6. code **KL early stopping.** ⊠ Instead of clipping, an implementation can stop reusing a batch once the policy has moved far enough. Spinning Up's PPO computes the approximate KL before every pass and ends the passes on a batch once it exceeds 1.5 times `target_kl`, which defaults to 0.01.

   1. Replace the `ppo_epochs` call in `train_ppo` by a loop of single passes, `ppo_epochs(..., 1, ...)`, as the loop that collects `rhos` in Section 15.2.4.5 does. Before each pass, compute the approximate KL from `log_prob_np` and apply the stopping rule. Record the number of passes spent on each batch.

2. Run the unclipped variant with early stopping on the eight seeds of the ablation. Does the stopping rule alone prevent the collapse? How does the number of passes per batch change over training?
3. Run the clipped variant with early stopping as well. Which of the two mechanisms limits the policy change of a batch in this run? What does the answer say about describing the clip as a trust region?

Adapted from Joshua Achiam, Spinning Up in Deep RL, PPO implementation, early stopping on `target_kl`.

7. code **The width of the band.** The clip has one hyperparameter, ϵ_{clip}, and the section never varied it. Rerun the clipped arm of Section 15.2.4.4, twenty epochs at the same learning rate on the same eight seeds, with $\epsilon_{\text{clip}} \in \{0.05, 0.1, 0.2, 0.5, 10\}$.
   1. For each width, count the seeds that hold the task and those that end dead, and record the environment steps to the first batch at 475.
   2. Which width behaves like the unclipped run of the ablation? What does that say about the ratios the unclipped run reaches?
   3. Plot the within-batch fraction of ratios outside the band for each width, as in Section 15.2.4.5. How do the narrowest bands trade speed against safety, and how does the fraction outside the band change with the width?

8. code **The standard recipe.** ▯▯ Build the loop that Section 15.2.4.6 describes: eight environments in a `gym.vector.SyncVectorEnv` stepped for 32 steps per update, GAE over the 32×8 rectangle with the critic's value at each cut and the autoreset step masked out, then four epochs over the rectangle in shuffled minibatches of 64, each minibatch taking one `ppo_epochs` pass with its own standardized advantages and one `fit_value` step toward the λ-return target computed before the epochs. Use a learning rate of 10^{-3} for both networks.
   1. Train three seeds and record the environment steps until the last eight completed episodes average at least 475. A correct implementation needs about 45,000 steps, between 35,000 and 55,000 on most seeds: more than the four-epoch arm of the reuse experiment in Section 15.2.4.3, which needs about 25,000.
   2. Raise the critic's learning rate to 10^{-2}, the rate of our loop, and keep the actor's at 10^{-3}. A correct implementation now reaches 475 in about 15,000 to 20,000 steps, fewer than any arm of the reuse experiment. Which network limited the run of (a)? Our loop at the same two rates, without vectorization or minibatches, needs about 35,000 to 40,000 steps. What do the short rectangles change about how often the policy is updated per environment step?
   3. Remove the mask on the autoreset step. Does the run still learn? Which transitions does the unmasked version add to the critic's targets, and why does CartPole barely notice them?

9. **The monotonic-improvement bound.** ▯▯ This problem derives Equation 15.21 by the coupling argument of Schulman et al. (2015) . Take an infinite-horizon task with $\gamma < 1$ and write $\pi = \pi_{\theta_{\text{old}}}$, $\tilde{\pi} = \pi_\theta$, $A = A^{\text{old}}$, and $\bar{A}(s) = \sum_a \tilde{\pi}(a \mid s)\, A(s,a)$. Let $\Delta_{\text{TV}} = \max_s D_{\text{TV}}(\pi(\cdot \mid s), \tilde{\pi}(\cdot \mid s))$ with the total variation distance $D_{\text{TV}}(p,q) = \frac{1}{2}\sum_a |p(a) - q(a)|$.
   1. Show from Equation 15.19 that $J(\theta) - J(\theta_{\text{old}}) = \sum_t \gamma^t \mathbb{E}_{s_t \sim \tilde{\pi}}[\bar{A}(s_t)]$, and that the population surrogate Equation 15.20 is $\bar{L}(\theta) = \sum_t \gamma^t \mathbb{E}_{s_t \sim \pi}[\bar{A}(s_t)]$, where

$s_t \sim \pi$ denotes the state distribution at time t under π.

2. Two distributions at total variation distance Δ_{TV} can be coupled: there is a joint distribution of $(a, \tilde{a})$ with marginals $\pi(\cdot \mid s)$ and $\tilde{\pi}(\cdot \mid s)$ under which $P(a \neq \tilde{a}) \leq \Delta_{\text{TV}}$ at every state. Use $\mathbb{E}_{a\sim\pi}[A(s,a)] = 0$ to show that $|\bar{A}(s)| \leq 2\Delta_{\text{TV}} A_{\max}$.
3. Run both policies from the same start state with coupled actions at every step, and let m_t count the steps before t at which the two actions differ. On the event $m_t = 0$, the two trajectories coincide up to time t. Show that $\left|\mathbb{E}_{s_t\sim\tilde{\pi}}[\bar{A}(s_t)] - \mathbb{E}_{s_t\sim\pi}[\bar{A}(s_t)]\right| \leq 4\Delta_{\text{TV}}(1-(1-\Delta_{\text{TV}})^t)A_{\max}$.
4. Sum over t to obtain

$$\left|J(\theta) - J(\theta_{\text{old}}) - \bar{L}(\theta)\right| \leq \frac{4\Delta_{\text{TV}}^2 \gamma A_{\max}}{(1-\gamma)^2}, \tag{15.24}$$

and finish the proof with $D_{\text{TV}}^2 \leq D_{\text{KL}}$.

5. Evaluate the coefficient $4\gamma/(1-\gamma)^2$ at $\gamma = 0.99$. Use (b) to show that $\bar{L}(\theta) \leq 2\Delta_{\text{TV}} A_{\max}/(1-\gamma)$. With Pinsker's inequality $D_{\text{KL}} \geq 2D_{\text{TV}}^2$, show that the bound certifies an improvement only if $\Delta_{\text{TV}} < (1-\gamma)/(4\gamma)$. How large can $\bar{L}(\theta)$ of a certified step then be, in units of $A_{\max}$? Compare this limit on Δ_{TV} with the total variation $\sqrt{\Delta_{\text{KL}}/2}$ that Pinsker's inequality permits at a state whose divergence equals the budget $\Delta_{\text{KL}} = 0.01$ that Schulman et al. (2015) use in all their experiments.

10. [extended] **TRPO from the pieces.** ▯▯ Replace the `ppo_epochs` call in `train_ppo` by one TRPO update (Schulman et al. 2015) on the constrained problem Equation 15.22 with $\Delta_{\text{KL}} = 0.01$, and leave the critic and the advantages unchanged. Treat the parameters of `ac.policy` as one flat vector, and freeze the collecting policy's action probabilities, `action_probs(ac, b.obs)`, together with `logp_old`.

    1. Compute the gradient $\mathbf{g}$ of the surrogate Equation 15.18 at θ_{old}. Write a function that returns the Fisher-vector product $F\mathbf{v}$, the Hessian of the mean divergence over the batch's states at θ_{old} applied to $\mathbf{v}$, by differentiating the gradient of that divergence a second time. Check it against a finite difference of the gradient.
    2. Solve $F\mathbf{x} = \mathbf{g}$ approximately by ten iterations of conjugate gradient, and scale the step to $\xi\mathbf{x}$ with $\xi = \sqrt{2\Delta_{\text{KL}}/\mathbf{x}^\top F\mathbf{x}}$. Halve ξ until the measured mean divergence is at most Δ_{KL} and the surrogate has increased.
    3. Train three seeds and compare them with the clipped runs of Section 15.2.4.4: learning curves, the measured divergence of each update, and the number of halvings. How many gradient and Fisher-vector computations does each method spend per batch?
    4. Engstrom et al. (2020) found that PPO and TRPO perform similarly when their code-level choices are matched. Here the two share every choice except the policy update. Does your comparison agree?

15.3 KL-Regularized Policy Optimization

Every method of Chapter 14 and of this chapter so far assumed that someone had written down the reward. For a frozen lake or a cart with a pole that is easy: reach the goal, keep the pole upright. For most tasks people care about it is not. Nobody can write a formula that scores a helpful answer, a safe lane change or a readable proof. People can, however, look at two attempts and say which one they prefer, and a reward can be fitted to such comparisons. This is how the language models in daily use were taught to be helpful: a reward model was fitted to human comparisons, and a policy was then trained against it (Christiano et al. 2017; Ouyang et al. 2022).

A fitted reward raises a question that a written one never did. The model is accurate only where its comparison data provide information, and it guesses elsewhere, while an optimizer searches everywhere. If the optimizer finds a region that the comparisons never covered and where the fitted reward happens to be high, it goes there, and the true quality falls. The remedy that practice settled on is to charge the policy for moving away from the policy that generated the comparisons, by a penalty on the KL divergence from that reference. The trust region of Section 15.2 limited how far each update moves from the previous policy. This penalty is part of the objective instead, so it changes the answer and not only the route to it, and the answer has a closed form: the reference distribution reweighted by the exponential of the reward.

We first build a gridworld whose true reward we know, fit a reward from comparisons, and watch an optimizer exploit its gaps. We then solve the penalized problem for one decision, extend the solution to many steps, and trace how the true quality rises and then falls as the policy moves away from its reference.

15.3.1 Learning a Reward from Comparisons

To see what goes wrong we need a task whose true reward we know, so that we can hide it from the learner and still grade the result. A small gridworld provides one.

15.3.1.1 The Gridworld and Its Reference Policy

The world is a 4×4 grid with deterministic moves, shown in Figure 15.6. An episode starts in the top-left cell and ends when it enters the goal in the bottom-right corner, which pays $+1$. Every other move costs 0.04, and a move into either of the two hazard cells side by side in the second row, the *hazard lane*, costs 1.5. The true reward $r^*(s, a)$ is the payment for the cell that action a enters from state s. The cells are numbered row by row, so the start is state 0, the hazards are states 5 and 6, and the goal is state 15. The actions 0 to 3 move left, down, right and up, and a move into the wall leaves the agent where it is.

The comparisons will be collected from trajectories of one fixed policy, the *reference policy* π_{ref}. We make it competent but hesitant: a softmax of the optimal action values at temperature 0.25, $\pi_{\text{ref}}(a \mid s) \propto e^{Q^*(s,a)/0.25}$. At the start, the two moves toward the goal have $Q^* = 0.59$ and the two into the walls 0.52, so the reference takes each good move with probability 0.28 and each wasted one with 0.22. A move into a hazard is worth about 1.5 less than the alternatives, which the softmax turns into a probability of about e^{-6}, so the reference rarely goes there. The cell builds the arrays $P[s, a, s']$ and $r^*[s, a]$ as a `d2l.TabularMDP` (Section 14.1), computes Q^* by running `d2l.value_iteration` and applying one more `backup` (Section 14.2), and forms the reference:

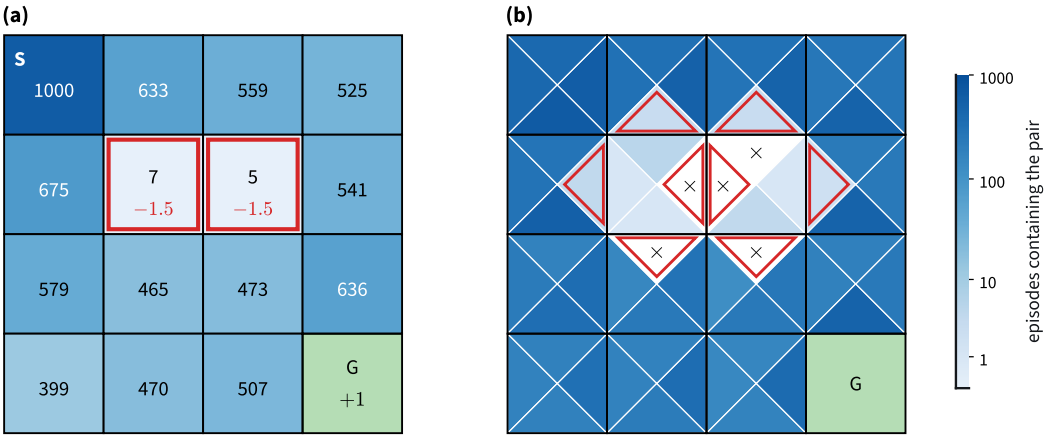


Figure 15.6: The gridworld of this section. (a) Episodes start in state 0 (S) and end in the goal, state 15 (G, $+1$); a move into a hazard cell, 5 or 6, costs 1.5 and every other move 0.04. Each cell shows in how many of the 1,000 reference trajectories it is visited. (b) The support of every state-action pair: each cell is split into four triangles for the moves left, down, right and up, shaded by the number of trajectories that contain the pair (log scale); a cross marks a pair that never occurs, and the eight moves into the hazard lane are outlined.

```
gamma = 0.95                                  # a 4 x 4 gridworld
P = np.zeros((16, 4, 16))
for s, a in np.ndindex(16, 4):
    x, y = s % 4, s // 4
    dx, dy = [(-1, 0), (0, 1), (1, 0), (0, -1)][a]   # left, down, right, up
    P[s, a, min(3, max(0, y + dy)) * 4 + min(3, max(0, x + dx))] = 1.0
P[15], P[15, :, 15] = 0.0, 1.0                # the goal absorbs
bonus = np.full(16, -0.04)                    # every step costs a little
bonus[[5, 6]], bonus[15] = -1.5, 1.0          # a hazard lane; the goal pays
r_true = np.einsum('sat,t->sa', P, bonus)     # r*(s, a): the entered bonus
r_true[15] = 0.0
mdp = d2l.TabularMDP(P, r_true, gamma)
Q_true = mdp.backup(d2l.value_iteration(mdp, 200)[-1])
pi_ref = np.exp(Q_true / 0.25)                # competent but hedging
pi_ref /= pi_ref.sum(1, keepdims=True)
```

15.3.1.2 Preferences and the Bradley-Terry Model

A comparison tells us which of two trajectories a judge preferred, not by how much. The Bradley-Terry model (Bradley and Terry 1952) turns such answers into a statistical model. It assigns each candidate a scalar score, here the return $r(\tau)$ of a trajectory under the reward r, and treats a comparison as a noisy observation of the score difference:

$$P(\tau \succ \tau') = \sigma\big(r(\tau) - r(\tau')\big), \qquad \sigma(u) = \frac{1}{1+e^{-u}}. \tag{15.25}$$

Thus two trajectories with equal returns are preferred either way with probability one half,

and a difference of one unit of return makes the better one win about three times in four.

In the gridworld the return is linear in the reward. Let $\mathbf{c}(\tau)$ be the discounted visit counts of a trajectory, one entry per state-action pair: entry (s, a) adds γ^t for every step t at which the trajectory takes a in s. Then $r(\tau) = \mathbf{c}(\tau)^\top r$, with the reward flattened into 64 entries, and fitting Equation 15.25 is logistic regression with features $\mathbf{c}(\tau) - \mathbf{c}(\tau')$ and the reward as weights. `sample_traj` returns $\mathbf{c}(\tau)$ for one trajectory of π_{ref}, which ends at the goal or after 40 steps, and `X` stacks 1000 of them. We label 6000 random pairs by coin flips with the probabilities of Equation 15.25 under r^* and fit $\hat{r}$ by gradient ascent on the log-likelihood, with a small weight decay, 0.02, so that the fit is well defined on pairs with little data. `episodes` counts the episodes in which each pair occurs, `core` marks the pairs that occur in at least 50, and `into` lists the four moves into the hazard lane from above and from the sides:

```
rng = np.random.default_rng(0)
sigmoid = lambda u: 1 / (1 + np.exp(-u))

def sample_traj(mdp, pi, rng, T=40):
    """Discounted visit counts c(tau) of one episode from state 0."""
    s, c = 0, np.zeros((16, 4))
    for t in range(T):
        a = rng.choice(4, p=pi[s])
        c[s, a] += mdp.gamma ** t              # the return is c(tau) . r
        s = mdp.P[s, a].argmax()
        if s == 15:
            break
    return c.ravel()

X = np.stack([sample_traj(mdp, pi_ref, rng) for _ in range(1000)])
R = X @ r_true.ravel()                         # true returns; the fit never sees them
i, j = rng.integers(0, 1000, (2, 6000))
y = (rng.random(6000) < sigmoid(R[i] - R[j])).astype(float)
D = X[i] - X[j]
lr, wd, num_steps = 0.5, 0.02, 4000            # ascent step, weight decay
w = np.zeros(64)
for _ in range(num_steps):                     # logistic regression on (D, y)
    w += lr * (D.T @ (y - sigmoid(D @ w)) / len(y) - wd * w)
r_hat = w.reshape(16, 4)
episodes = (X > 0).sum(0).reshape(16, 4)       # episodes containing each pair
core = episodes >= 50
into = ([1, 4, 2, 7], [1, 2, 1, 0])            # into cells 5, 6 from above, sides
print(f'never visited: {(episodes == 0).sum()} pairs; '
      f'in at least 50 episodes: {core.sum()} pairs')
print(f'rms fit error on those {core.sum()}: '
      f'{np.sqrt(((r_hat - r_true)[core] ** 2).mean()):.3f}')
print(f'moves into the hazards: in {episodes[into]} episodes, '
      f'fitted {np.round(r_hat[into], 2)}')
```

```
never visited: 9 pairs; in at least 50 episodes: 46 pairs
rms fit error on those 46: 0.193
moves into the hazards: in [3 4 3 2] episodes, fitted [-0.03 -0.05 -0.02 -0.02]
```

The fit has an rms error of 0.19 on the forty-six pairs of `core`. Nine pairs never occur: the four actions at the goal, where every episode has already ended, and the five marked with

crosses in Figure 15.6. Their fitted rewards stay at the initial value of zero. The moves into the hazard lane from above and from the sides occur in two to four of the thousand episodes each, and the two from below never occur. That is too few to move a fitted reward far from zero, where the weights started and where the weight decay pulls them; truly each costs 1.5. A neural reward model would instead extrapolate according to its inductive bias, for better or worse (exercise 5).

15.3.1.3 Invariance to Shaping

An rms error of 0.19 against step costs of 0.04 looks large, but most of it is invisible to any comparison. Potential-based shaping, introduced in Section 14.1, adds $\gamma\Phi(s') - \Phi(s)$ to the reward of each move from s to s', for a function Φ on the states with $\Phi = 0$ at the goal, and Section 14.2 proved that it leaves the optimal policy unchanged. Along a trajectory these terms telescope, so every trajectory from state 0 that reaches the goal changes its return by the same amount, $-\Phi(0)$. Every comparison involves two trajectories from state 0, so the shift cancels in $r(\tau) - r(\tau')$, and Equation 15.25 cannot detect it. (The one trajectory in ten that is cut off after forty steps keeps a final term $\gamma^{40}\Phi(s_{40})$, too small to pin Φ down.) The comparisons therefore determine the reward only up to shaping, and among the rewards they cannot tell apart, the weight decay prefers the one with the smallest weights.

The cell finds the potential whose shaping term best explains the fit's error on `core`, in the least-squares sense. Row (s, a) of `F` multiplies Φ to give $\gamma \sum_{s'} P(s' \mid s, a)\, \Phi(s') - \Phi(s)$, and dropping the last column sets $\Phi(15) = 0$:

```
err = (r_hat - r_true)[core]
F = (gamma * P - np.eye(16)[:, None, :])[core][:, :15]  # shaping per unit of Phi
Phi = np.linalg.lstsq(F, err, rcond=None)[0]
print(f'rms error on core {np.sqrt((err ** 2).mean()):.3f}, after removing '
      f'the best shaping term {np.sqrt(((err - F @ Phi) ** 2).mean()):.3f}')
print(f'Phi(0) = {Phi[0]:.2f}; moves 13 -> 14 -> goal: fitted '
      f'{r_hat[13, 2]:+.2f}, {r_hat[14, 2]:+.2f}, true '
      f'{r_true[13, 2]:+.2f}, {r_true[14, 2]:+.2f}')
```

```
rms error on core 0.193, after removing the best shaping term 0.032
Phi(0) = 0.78; moves 13 -> 14 -> goal: fitted -0.08, +0.09, true -0.04, +1.00
```

Removing the shaping term leaves an rms error of 0.03: up to shaping, the fit is good where the data are. The potential rises from about 0.8 at the start to about 0.9 beside the goal; at the hazard cells, which no pair of `core` touches, it is left undetermined. Such a potential subtracts about 0.9 from the move into the goal and about $(1 - \gamma) \times 0.8 = 0.04$ from every other move, which is what the printed rewards show: the last move pays 0.09 instead of 1, and an ordinary move costs 0.08 instead of 0.04. Both rewards order every pair of trajectories from state 0 alike, and the fitted one has the smaller weights. Under it, the hazard moves, estimated near zero, are cheaper than an ordinary move.

Language models face the same invariance. A comparison there involves two responses y and y' to the same prompt x, and a reward $r(x, y) + f(x)$, for any function f of the prompt, predicts every comparison as $r(x, y)$ does, so a learned reward is identified only relative to each prompt (exercise 1). Policy-gradient methods are unaffected: subtracting a function of the prompt from the reward is a baseline, which leaves the expected gradient unchanged (Section 14.7). The group mean of GRPO is such a baseline, except that it includes the response's own reward, which scales the expected update by $(K - 1)/K$ for groups of K

responses, as that section showed.

15.3.2 Optimizing a Fitted Reward

We now do what an agent would do with the fitted reward: find the policy that maximizes it. With the dynamics known, value iteration finds it. `plan` returns the optimal deterministic policy for a reward as one-hot rows. `ret` returns the exact return of a policy from state 0 through `policy_value` of Section 14.2, which solves the linear Bellman equations $V^\pi = (I - \gamma P_\pi)^{-1} r_\pi$ in one step. We plan once on the true and once on the fitted reward, and grade both plans under both rewards:

```
def plan(r):
    """Optimal deterministic policy (one-hot rows) for reward r."""
    m = d2l.TabularMDP(P, r, gamma)
    return np.eye(4)[m.backup(d2l.value_iteration(m, 200)[-1]).argmax(1)]

def ret(pi, r):
    """Exact return of policy pi from state 0 under reward r."""
    return d2l.TabularMDP(P, r, gamma).policy_value(pi)[0]

for name, r in (('true reward  ', r_true), ('fitted reward', r_hat)):
    pi = plan(r)
    print(f'plan on {name}: fitted return {ret(pi, r_hat):+.2f}, '
          f'true return {ret(pi, r_true):+.2f}')
```

```
plan on true reward  : fitted return -0.24, true return +0.59
plan on fitted reward: fitted return -0.03, true return -2.11
```

Under the fitted reward the plan on $\hat{r}$ scores -0.03, above the -0.24 of the true optimum. The true optimum's fitted score lies 0.83 below its true score, close to the shaping offset $\Phi(0) = 0.78$, so only differences between fitted scores mean anything. Under the true reward the two plans score -2.11 and $+0.59$. The plan on $\hat{r}$ walks $0 \to 1 \to 5 \to 6 \to 7 \to 11 \to 15$, straight through both hazard cells, a route as short as the safe one along the border and cheaper under $\hat{r}$. Value iteration optimized the reward it was given; the failure lies in the reward outside the reference's data. Over 24 preference datasets drawn with different seeds, the plan on $\hat{r}$ crosses the hazard cells in 19 and never reaches the goal in 4 more; one dataset yields the true optimum.

This failure has a name. Goodhart's law states that a measure loses its value once it becomes a target. In reinforcement learning, a policy that scores highly under a specified or fitted reward while defeating the intended objective is said to *hack* the reward, and many cases are catalogued in (Krakovna et al. 2020). The approach bonus of Section 14.2, which made the optimal policy circle forever, was the same mechanism with a hand-written error. Here the error is statistical, and it sits where the data are thin.

15.3.3 The KL-Regularized Objective

The fitted reward deserves trust only where the reference policy went. A policy that stays close to the reference stays where the data are, so we charge for moving away from it. KL divergence is the natural measure here: it is an expectation under the policy itself, so it can be estimated from the policy's own samples, and it is infinite for a policy that takes an action

the reference never takes. For a single decision with reward $r(a)$ over a finite set of actions, a reference π_{ref} and a coefficient $\beta > 0$, the *KL-regularized objective* is

$$\max_{\pi} \; \mathbb{E}_{a\sim\pi}[r(a)] - \beta\, D_{\text{KL}}(\pi\|\pi_{\text{ref}}). \tag{15.26}$$

The first term is the expected reward, and the second subtracts β units of reward for every nat of divergence from the reference. The coefficient β is the weight of the penalty; the reference stays fixed.

15.3.3.1 The Closed-Form Optimum for One Decision

Proposition. For $\beta > 0$, the objective Equation 15.26 is maximized by the unique policy

$$\pi^\star(a) = \frac{1}{Z}\,\pi_{\text{ref}}(a)\, e^{r(a)/\beta}, \qquad Z = \sum_a \pi_{\text{ref}}(a)\, e^{r(a)/\beta}, \tag{15.27}$$

and the maximal value is $\beta \log Z$.

Proof. By construction $r(a) = \beta \log\left(\pi^\star(a) Z / \pi_{\text{ref}}(a)\right)$. Substituting into the objective, for any policy π,

$$\mathbb{E}_\pi[r] - \beta D_{\text{KL}}(\pi\|\pi_{\text{ref}}) = \sum_a \pi(a)\,\beta \log \frac{\pi^\star(a)\, Z}{\pi(a)} = \beta \log Z - \beta\, D_{\text{KL}}(\pi\|\pi^\star).$$

Gibbs' inequality, proved in Chapter 37, says the remaining divergence is nonnegative and zero exactly when $\pi = \pi^\star$. ■

Thus the optimum multiplies the reference by an exponential tilt $e^{r/\beta}$ and renormalizes, and the value it achieves is a log-partition function. Figure 15.7 draws the product for five actions. The reference there is deliberately not monotone in the reward, and its dip at the second action survives a mild tilt: the optimum reweights the reference rather than ranking the actions by reward.

Two boundary cases complete the statement. First, the optimum inherits the reference's support. If $\pi_{\text{ref}}(a) = 0$, any policy that puts mass on a has infinite divergence, and Equation 15.27 gives a probability zero, whatever its reward. Second, as $\beta \to 0$ the optimum concentrates on the best action in the support of π_{ref}; if several actions tie for best, the limit is the reference renormalized over the tied set.

We check the proposition numerically by solving Equation 15.26 with an iterative method that knows nothing of the closed form. `solve_kl` runs exponentiated gradient ascent: it multiplies the current policy by the exponential of a step size times the gradient, $r - \beta(\log(\pi/\pi_{\text{ref}}) + 1)$, and renormalizes onto the simplex. The $+1$ is the same for every action and cancels in the normalization, so the code drops it:

```
p_ref = np.array([0.30, 0.10, 0.25, 0.20, 0.15])
r = np.array([0.0, 0.5, 1.0, 2.0, 3.0])

def solve_kl(r, p, beta, steps=3000, eta=0.3):
    pi = np.full_like(p, 1 / len(p))
```

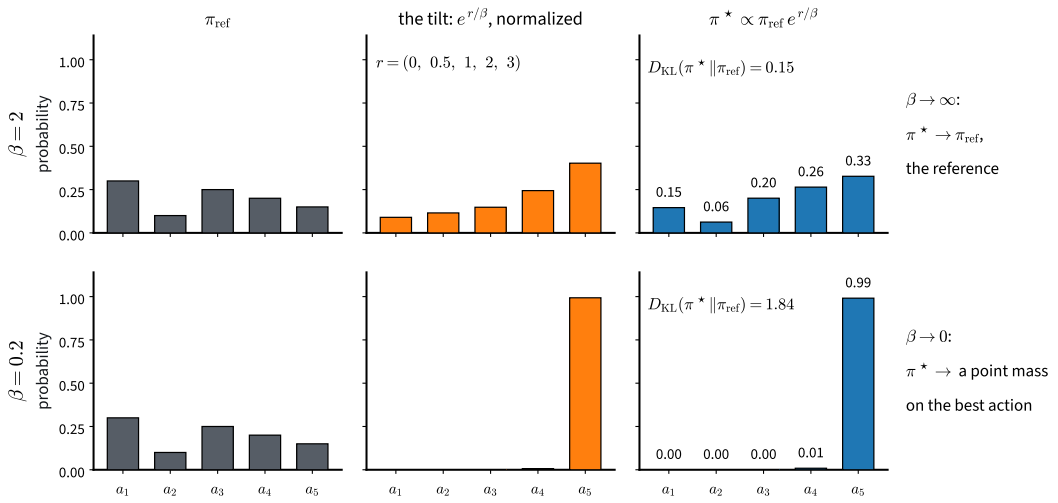


Figure 15.7: The optimum of the penalized objective as a product. Each row multiplies the reference $\pi_{\text{ref}} = (0.30,\ 0.10,\ 0.25,\ 0.20,\ 0.15)$ by the tilt $e^{r/\beta}$ for the reward $r = (0,\ 0.5,\ 1,\ 2,\ 3)$ and renormalizes, giving $\pi^\star$; the top row uses $\beta = 2$, the bottom $\beta = 0.2$. The reference's dip at a_2 survives into $\pi^\star$ in the top row and is overwhelmed in the bottom row. The margins annotate the two limits: $\beta \to \infty$ returns the reference, $\beta \to 0$ a point mass on the best action.

```
    for _ in range(steps):                    # exponentiated gradient ascent
        pi = pi * np.exp(eta * (r - beta * np.log(pi / p)))
        pi /= pi.sum()
    return pi

for beta in (5.0, 2.0, 0.5, 0.2):
    pi = solve_kl(r, p_ref, beta)
    print(f'beta = {beta:3}: pi_beta = {np.round(pi, 3)}, '
          f'reward {pi @ r:.2f}, KL {(pi * np.log(pi / p_ref)).sum():.2f}')
```

```
beta = 5.0: pi_beta = [0.233 0.086 0.237 0.232 0.212], reward 1.38, KL 0.02
beta = 2.0: pi_beta = [0.146 0.062 0.2   0.264 0.327], reward 1.74, KL 0.15
beta = 0.5: pi_beta = [0.004 0.004 0.025 0.148 0.819], reward 2.78, KL 1.26
beta = 0.2: pi_beta = [0.    0.    0.    0.009 0.991], reward 2.99, KL 1.84
```

As β falls from 5 to 0.2 the expected reward rises from 1.38 to 2.99 and the divergence from 0.02 to 1.84 nats. The two rows of Figure 15.7 are the optima at $\beta = 2$, with divergence 0.15 nats, and at $\beta = 0.2$. We now compare each converged iterate with Equation 15.27, and the achieved objective with $\beta \log Z$. `pi_star` evaluates the closed form, subtracting the largest reward inside the exponential so that it cannot overflow; the shift cancels in the normalization:

```
def pi_star(r, p, beta):
    star = p * np.exp((r - r.max()) / beta)    # eq_kl_optimum, stabilized
    return star / star.sum()
```

```
gap = max(np.abs(solve_kl(r, p_ref, b) - pi_star(r, p_ref, b)).max()
          for b in (5.0, 2.0, 0.5, 0.2))
pi2 = solve_kl(r, p_ref, 2.0)
print(f'largest gap to the closed form over the ladder: {gap:.1e}')
print(f'objective at beta = 2: numerical '
      f'{pi2 @ r - 2.0 * (pi2 * np.log(pi2 / p_ref)).sum():.6f}, '
      f'beta log Z = {2.0 * np.log(p_ref @ np.exp(r / 2.0)):.6f}')
```

```
largest gap to the closed form over the ladder: 1.4e-16
objective at beta = 2: numerical 1.442004, beta log Z = 1.442004
```

The iterates agree with the closed form to floating-point resolution at every β, and the objective matches $\beta \log Z$ to six decimal places.

15.3.3.2 Four Consequences of the Closed Form

Without the penalty, the optimum is a point mass. Letting $\beta \to 0$ in Equation 15.27 sends all mass to the highest-reward action, or, under ties, to the reference's distribution over the tied set. This is the unregularized problem, whose solution is the argmax. The statement concerns the optimum, not the course of a particular training run. The entropy of the PPO policy in Section 15.2 fell during training, which is consistent with movement toward the near-deterministic optimum of PPO's lightly entropy-regularized objective. A penalty with a larger weight makes a stochastic policy optimal.

The coefficient interpolates between the reference and the argmax. The closed form at a very large and a very small β:

```
for beta in (100.0, 0.01):
    print(f'beta = {beta:6}: pi_star = {np.round(pi_star(r, p_ref, beta), 3)}')
```

```
beta =  100.0: pi_star = [0.297 0.099 0.25  0.202 0.153]
beta =   0.01: pi_star = [0. 0. 0. 0. 1.]
```

Large β reproduces the reference, small β is greedy, and every policy in between is the exact optimum for some value of β.

A uniform reference turns the penalty into an entropy bonus. Against a uniform reference, $D_{\mathrm{KL}}(\pi \| \text{uniform}) = \log|\mathcal{A}| - H(\pi)$, so the penalty is a bonus $\beta H(\pi)$ for entropy plus a constant, and Equation 15.26 becomes the *maximum-entropy* objective. The closed form specializes to a softmax of the reward:

```
boltzmann = np.exp(r / 0.5) / np.exp(r / 0.5).sum()
print(np.allclose(pi_star(r, np.full(5, 0.2), 0.5), boltzmann))
```

```
True
```

This is the form of the softmax policies of Section 14.6. The Boltzmann exploration rule of Section 14.4, with β as its temperature, is therefore the exact optimum of a one-step objective with an entropy bonus.

The optimum is a posterior. The closed form Equation 15.27 has the form of Bayes' rule: π_{ref} is a prior, $e^{r/\beta}$ an unnormalized likelihood, $\pi^\star$ the posterior and Z the normalizer.

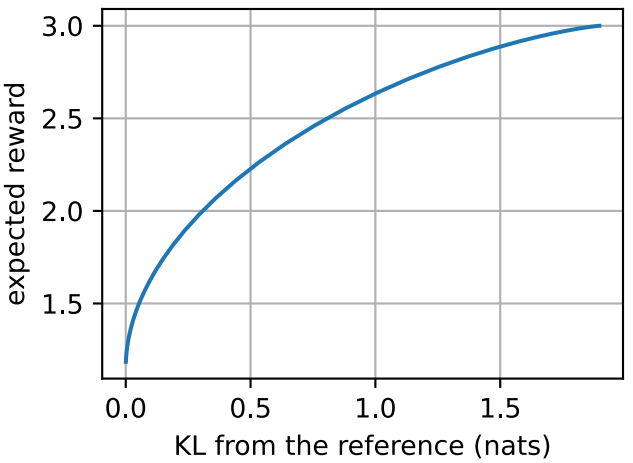
Control-as-inference methods build on this identity, reading rewards as log-likelihoods of an event called optimality and regularized policy optimization as variational inference (Levine 2018; Korbak et al. 2022).

15.3.3.3 The One-Decision Frontier

Each β gives one optimum, with an expected reward and a divergence from the reference. Sweeping β traces the best reward attainable at each divergence, the *reward-divergence frontier*:

```
front = np.stack([pi_star(r, p_ref, b) for b in np.logspace(1.5, -1.5, 61)])
kl_f = (front * np.log(front / p_ref)).sum(1)
print(f'KL 0 pays {p_ref @ r:.2f}; the frontier ends at KL '
      f'{np.log(1 / p_ref[r.argmax()]):.2f}, reward {r.max():.0f}')
d2l.plot(kl_f, front @ r, 'KL from the reference (nats)', 'expected reward')
```

```
KL 0 pays 1.15; the frontier ends at KL 1.90, reward 3
```



The frontier starts at the reference's average reward, 1.15, and ends at the point mass on the best action, whose divergence is $\log(1/0.15) = 1.90$ nats. It is increasing and concave, with slope β at the optimum for β (exercise 7), so each further nat yields less reward. The two rows of Figure 15.7 are two points on this curve. Best-of-N sampling, which draws N answers from the reference and keeps the one with the highest reward, moves along the same axis without any training: its divergence from the reference is at most $\log N - (N-1)/N$, about 1.8 nats for $N = 16$ (Gao et al. 2023). On the gridworld we will trace the same kind of curve, but grade it with the true reward instead of the reward being optimized.

15.3.3.4 Trust Region versus Penalty

Trust regions and the fixed-reference penalty both use KL divergence, in different roles.

> **Two uses of KL divergence.** A **trust region** (Section 15.2) measures divergence from the *previous iterate*. It limits each update and protects a local approximation, and its reference moves after every update. A **fixed-reference penalty** (this section) is part of the objective and moves the optimum to Equation 15.27.

PPO-style fine-tuning of language models uses both: a clipped surrogate relative to the sampling policy for stable updates, and a reward penalty relative to the frozen initial model to control the final policy (Ouyang et al. 2022).

The closed form shows where a trust region alone leads. Suppose every update solves its trust-region problem exactly, with a weight $\tilde{\beta}$ on the divergence from the previous iterate, $\pi_{k+1} = \operatorname{argmax}_\pi \mathbb{E}_\pi[r] - \tilde{\beta} D_{\mathrm{KL}}(\pi \| \pi_k)$, starting from $\pi_0 = \pi_{\mathrm{ref}}$. By the proposition each step multiplies the policy by $e^{r/\tilde{\beta}}$, so after k steps

$$\pi_k(a) = \frac{\pi_{\mathrm{ref}}(a)\, e^{k\, r(a)/\tilde{\beta}}}{\sum_{a'} \pi_{\mathrm{ref}}(a')\, e^{k\, r(a')/\tilde{\beta}}}, \tag{15.28}$$

the closed form with coefficient $\tilde{\beta}/k$. As k grows the coefficient goes to zero, and the iterates approach the point mass on the best action. A trust region shapes the path but not the destination; only a fixed reference changes where the iterates converge (exercise 8).

15.3.3.5 The Direction of the Divergence

The penalty $D_{\mathrm{KL}}(\pi \| \pi_{\mathrm{ref}})$ is the *reverse* divergence in the taxonomy of Section 37.3.2, an expectation under π that can be estimated from the policy's own samples. The closed form shows how it treats an action a whose reward falls short of the best by $\Delta(a) > 0$: relative to the best action, its probability under $\pi^\star$ is its reference probability times $e^{-\Delta(a)/\beta}$. The forward penalty $D_{\mathrm{KL}}(\pi_{\mathrm{ref}} \| \pi)$ is infinite for a policy that drops any action the reference takes, and its optimum keeps every action at a floor of order $\beta/\Delta(a)$ times its reference probability (exercise 9). Both directions keep every action the reference takes at positive probability for every $\beta > 0$. They differ in how fast poor actions fade as β shrinks: exponentially in $1/\beta$ under the reverse penalty, only linearly in β under the forward one.

15.3.4 The Soft Backup

The gridworld is not a single decision. In an MDP the penalty is charged at every step, and the objective from state 0 becomes

$$\mathbb{E}_\pi\Big[\sum_{t\geq 0} \gamma^t \big(r(s_t, a_t) - \beta\, D_{\mathrm{KL}}\big(\pi(\cdot \mid s_t) \| \pi_{\mathrm{ref}}(\cdot \mid s_t)\big)\big)\Big]. \tag{15.29}$$

At each state the policy faces a one-decision problem of the kind just solved. The question is which reward that decision should be tilted by.

15.3.4.1 One Step before the End

At the last decision of an episode an action's worth is its reward, and the proposition applies verbatim: the optimum there is the reference tilted by $e^{r/\beta}$, and the state is worth $\beta \log Z$. One step earlier, an action's worth is its reward plus the worth of the state it leads to. That worth is what the optimal policy *there* achieves under the penalty, $\beta \log Z$, which is less than the best reward available there. The optimal policy at the last step takes that best action only with some probability and is charged the penalty as well.

A two-step example makes the difference concrete. Let the five-action decision of Figure 15.7 with $\beta = 2$ be the last step of an episode; its value is $\beta \log Z = 1.44$, although its best reward is 3. One step earlier the agent chooses between entering that decision, which pays nothing at once, and stopping with a sure reward of 2; the reference picks each with probability one half, and there is no discount. The cell tilts the first decision once by the soft value 1.44 and once by the best reward 3, and evaluates the penalized objective of both two-step policies:

```
beta, stop = 2.0, 2.0                          # stopping pays 2 for sure
V_next = beta * np.log(p_ref @ np.exp(r / beta))   # value of the last step
for name, q in (('soft value', V_next), ('best reward', r.max())):
    pi0 = pi_star(np.array([q, stop]), np.array([0.5, 0.5]), beta)
    J = pi0 @ [V_next, stop] - beta * (pi0 * np.log(pi0 / 0.5)).sum()
    print(f'tilt by the {name} {q:.2f}: pi(enter) = {pi0[0]:.2f}, '
          f'objective {J:.3f}')
Z0 = 0.5 * np.exp(V_next / beta) + 0.5 * np.exp(stop / beta)
print(f'beta log Z of the first decision: {beta * np.log(Z0):.3f}')
```

```
tilt by the soft value 1.44: pi(enter) = 0.43, objective 1.740
tilt by the best reward 3.00: pi(enter) = 0.62, objective 1.592
beta log Z of the first decision: 1.740
```

Tilting by the soft value attains the optimum, $\beta \log Z$ of the first decision. Tilting by the best reward values the second decision at 3, although the penalized policy there earns only 1.44, so it enters too often and loses about 0.15. The tilt must use the value that accounts for every later reward *and* every later penalty.

15.3.4.2 Soft Value Iteration

Writing this value at every state gives the general form. Let $Q_\beta(s, a)$ be the worth of action a at state s under the penalized objective, and $V_\beta(s)$ the worth of state s. Applying the proposition at every state, with $Q_\beta(s, \cdot)$ in the role of the reward,

$$V_\beta(s) = \beta \log \sum_a \pi_{\text{ref}}(a \mid s)\, e^{Q_\beta(s,a)/\beta}, \qquad Q_\beta(s, a) = r(s, a) + \gamma \sum_{s'} P(s' \mid s, a)\, V_\beta(s'), \tag{15.30}$$

and the optimal policy is $\pi_\beta(a \mid s) \propto \pi_{\text{ref}}(a \mid s)\, e^{Q_\beta(s,a)/\beta}$. These are the Bellman optimality equations of Section 14.2 with the maximum over actions replaced by a reference-weighted log-sum-exp, a *soft maximum* that tends to the hard maximum over the reference's support as $\beta \to 0$. The hard maximum is biased upward when the action values are noisy estimates, as Section 14.3 demonstrated; the log-sum-exp, being convex, is biased upward too, only less sharply. With a uniform reference this is the soft value iteration of maximum-entropy reinforcement learning.

Value iteration solved the hard equations by sweeping $V \leftarrow \max_a \text{backup}(V)$ from $V = 0$. The same sweep with the soft maximum solves Equation 15.30. The proof that the hard sweep contracts (Section 14.2.2.3) used only two properties of the maximum: it is monotone, and adding a constant to every entry adds that constant to the maximum. The soft maximum has both, so the soft sweep is also a γ-contraction, for every $\beta > 0$ and every reference (exercise 4). `soft_max` computes the soft maximum of each row of an array of action values,

after subtracting the row's maximum for numerical stability, and returns the tilted policy with it. `soft_v` sweeps `backup` followed by `soft_max`:

```
def soft_max(Q, p, beta):
    """Row-wise beta log sum_a p(a) exp(Q(a) / beta), and the tilted policy."""
    m = Q.max(1, keepdims=True)                # subtracted for stability
    tilt = p * np.exp((Q - m) / beta)
    V = m[:, 0] + beta * np.log(tilt.sum(1))
    return V, tilt / tilt.sum(1, keepdims=True)

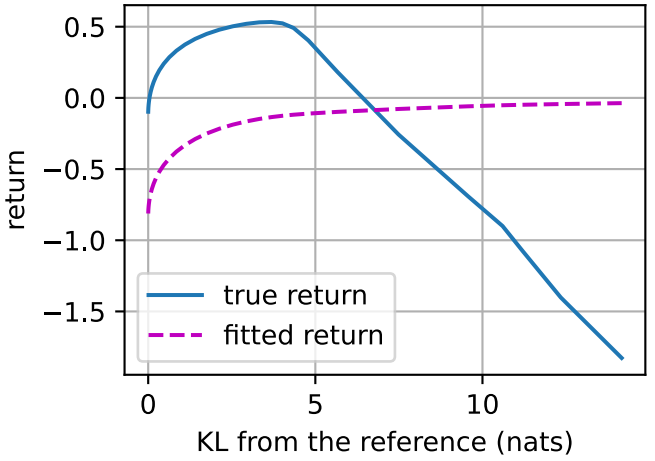
def soft_v(mdp, p, beta, num_iters=400):
    """Soft value iteration V <- soft_max(backup(V)); returns V and pi_beta."""
    V = np.zeros(mdp.num_states)
    for _ in range(num_iters):
        V, pi = soft_max(mdp.backup(V), p, beta)
    return V, pi
```

15.3.4.3 True Return against the KL Budget

We now trace the frontier on the gridworld, with one change: the optimizer sees the fitted reward $\hat{r}$, and we grade with the true one. For each of 36 values of β from 10 down to $10^{-2.5}$, `soft_v` computes the exact penalized optimum π_β for $\hat{r}$, and we record its true return, its fitted return and its divergence from the reference. For the divergence we use the penalty of the MDP objective, $\mathbb{E}_{\pi_\beta}\left[\sum_t \gamma^t D_{\mathrm{KL}}(\pi_\beta(\cdot \mid s_t) \| \pi_{\mathrm{ref}}(\cdot \mid s_t))\right]$ from state 0. It is the KL divergence between the distributions of trajectories that π_β and π_{ref} generate when every episode also ends with probability $1-\gamma$ after each step (exercise 2). The code weights the per-state divergences `kl` by `visits`, the expected discounted number of visits to each state from state 0, which one linear solve with the transition matrix of π_β provides. It also records the largest residual of Equation 15.30 over the sweep:

```
m_hat = d2l.TabularMDP(P, r_hat, gamma)
kls, true_ret, fit_ret, resid = [], [], [], 0.0
for beta in np.logspace(1, -2.5, 36):
    V, pi = soft_v(m_hat, pi_ref, beta)
    V_new, _ = soft_max(m_hat.backup(V), pi_ref, beta)
    resid = max(resid, np.abs(V_new - V).max())
    kl = (pi * np.log(np.where(pi > 0, pi / pi_ref, 1.0))).sum(1)
    P_pi = np.einsum('sa,sat->st', pi, P)
    visits = np.linalg.solve(np.eye(16) - gamma * P_pi.T, np.eye(16)[0])
    kls.append(visits @ kl)                    # discounted KL from state 0
    true_ret.append(ret(pi, r_true))
    fit_ret.append(ret(pi, r_hat))
k = int(np.argmax(true_ret))
print(f'largest soft Bellman residual: {resid:.0e}')
print(f'reference: true return {ret(pi_ref, r_true):+.2f}; beta = 10: '
      f'KL {kls[0]:.3f}, true return {true_ret[0]:+.2f}')
print(f'peak: true return {true_ret[k]:+.2f} at KL {kls[k]:.1f}; '
      f'smallest beta: {true_ret[-1]:+.2f} at KL {kls[-1]:.1f}')
d2l.plot(kls, [true_ret, fit_ret], 'KL from the reference (nats)', 'return',
         legend=['true return', 'fitted return'])
```

```
largest soft Bellman residual: 0e+00
reference: true return -0.12; beta = 10: KL 0.001, true return -0.10
peak: true return +0.53 at KL 3.7; smallest beta: -1.83 at KL 14.2
```



Every point in the plot is an exact optimum of the penalized objective; the largest Bellman residual is below 10^{-10}. The fitted return rises throughout, and for exact optima it must. If $\beta_1 < \beta_2$ and the optima have fitted returns J_1, J_2 and divergences D_1, D_2, optimality gives $J_1 - \beta_1 D_1 \geq J_2 - \beta_1 D_2$ and $J_2 - \beta_2 D_2 \geq J_1 - \beta_2 D_1$. Adding the two gives $(\beta_2 - \beta_1)(D_1 - D_2) \geq 0$, so $D_1 \geq D_2$, and the first inequality then gives $J_1 \geq J_2$.

The true return behaves differently. It rises from the reference's -0.12 to $+0.53$ at 3.7 nats, close to the true optimum of $+0.59$, and then falls to -1.83 as the policy concentrates on the hazard lane. A moderate push on the fitted reward improves the true return, and a stronger one exploits the moves the data never measured. Over 24 preference datasets the peak lies between about $+0.5$ and the optimum $+0.59$ at a few nats, and the true return falls after it in 21. In the other three it does not fall within the swept range: in one the fitted optimum is the true one, and in two the fitted values of the safe and hazard routes nearly tie, so the penalized optimum keeps the safe route down to $\beta = 10^{-2.5}$. How far from the reference the peak sits and how deep the fall goes depend on how well the comparisons cover the hazard and on what it costs. Gao et al. (2023) measured the same rise and fall for language models optimized against a learned reward model and graded by a held-out one.

The peak of the true return is invisible to the optimizer, which sees only the fitted curve, and that curve rises monotonically. In practice β is therefore chosen by evaluating the policy against something other than the reward model it was trained on: held-out comparisons, human judgments, or a checker. With a uniform reference the same penalty is an entropy bonus, and Section 15.5 builds an off-policy actor-critic for continuous actions that maximizes it. Its critic evaluates its own policy under the reward with the entropy bonus rather than iterating the log-sum-exp backup; by the identity in the proof of the proposition, the two agree when the policy is the tilted one. Section 15.8 applies the closed form Equation 15.27 to the verifier task of Chapter 14, where it predicts the β at which a hedged answer stops paying. Every method so far has learned from fresh samples of the current policy or from a known model; Section 15.4 asks whether a network can learn action values from stored transitions of any policy, as the Q-table of Section 14.3 did.

15.3.5 Summary

A reward fitted to comparisons is accurate only where the comparisons are. Comparisons between trajectories from a common start cannot see a potential shaping of the reward, and pairs the data never cover keep whatever value the fit started with. An optimizer exploits those gaps: in the gridworld, the plan on the fitted reward walks through hazard cells that the reference rarely entered. Penalizing the KL divergence from the reference changes the objective, and its exact optimum is the reference tilted by $e^{r/\beta}$, with value $\beta \log Z$; a uniform reference turns the penalty into an entropy bonus. In an MDP the tilt uses the soft action value, which satisfies a Bellman equation with a log-sum-exp in place of the maximum. Sweeping the penalty's weight, the fitted return rises monotonically while the true return peaks at a moderate divergence and then falls, so the weight must be chosen by an evaluation the optimizer does not see.

15.3.6 Exercises

1. **Identifiability and scale.** The Bradley-Terry model Equation 15.25 is fitted to comparisons of two responses y and y' to the same prompt x.
   1. Show that replacing $r(x, y)$ by $r(x, y) + f(x)$, for any function f of the prompt, leaves the likelihood of every such dataset unchanged. Conversely, join two responses to x by an edge whenever the dataset compares them. Show that two rewards that assign the same preference probability to every compared pair differ by a constant on each connected component of this graph. What is identified when the graph of one prompt has several components?
   2. Suppose that the annotators label according to $P(y \succ y') = \sigma(\omega\,(r^*(x, y) - r^*(x, y')))$ for some consistency $\omega > 0$, and that Equation 15.25 is fitted by maximum likelihood on unlimited data. Which reward does the fit recover?
   3. Show that the optimum Equation 15.27 for the fitted reward at coefficient β is the optimum for r^* at coefficient β/ω. Suppose that β was tuned against a reward model fitted to the labels of one pool of annotators. How strong a penalty is the same β for a reward model fitted to the labels of a less consistent pool, that is, one with a smaller ω?
2. code **Trajectory divergence.** Let $p_\pi(\tau)$ denote the probability that a policy π, started at state 0, produces the trajectory $\tau = (s_0, a_0, s_1, a_1, \dots, s_T)$ of a fixed horizon T.
   1. Show that

$$D_{\mathrm{KL}}(p_\pi \| p_{\pi_{\mathrm{ref}}}) = \mathbb{E}_\pi\Big[\sum_{t=0}^{T-1} D_{\mathrm{KL}}(\pi(\cdot \mid s_t) \| \pi_{\mathrm{ref}}(\cdot \mid s_t))\Big].$$

      Which factors of $p_\pi(\tau)$ cancel, and why can the divergence therefore be estimated from trajectories of π without a model of the transitions?
   2. Let the trajectory instead end after each step with probability $1 - \gamma$, independently of the actions. Show that the divergence between the two trajectory distributions becomes the discounted sum of per-state divergences that the sweep of Section 15.3.4.3 appends to `kls` as `visits @ kl`.
   3. Take the policy π_β at the peak of the true return in that sweep. Average $\mathbf{c}(\tau)^\top \log(\pi_\beta/\pi_{\mathrm{ref}})$, with the logarithm taken entrywise and flattened like $\mathbf{c}(\tau)$, over 2000 trajectories from `sample_traj(mdp, pi, rng, T=400)`, and compare the average with the divergence of 3.7 nats printed at the peak. Some of the per-trajectory values are negative. Why does this not contradict Gibbs' inequality?

3. code **Tilting by the unregularized value.** Section 15.3.4.2 tilts π_{ref} by the soft value Q_β. Tilting instead by the unregularized action value Q^* gives a different family of policies, which in general contains the optimum for no value of β.
   1. Compute Q^* for $\hat{r}$ by value iteration and form $\pi(a \mid s) \propto \pi_{\text{ref}}(a \mid s)\, e^{Q^*(s,a)/\beta}$ for every β of the sweep in Section 15.3.4.3. Plot the fitted return of this family against its discounted divergence from π_{ref}, together with `fit_ret` against `kls`.
   2. Why can no member of the heuristic family lie above the curve of the exact optima?
   3. In which limits of β do the two families approach each other, and why do they differ in between? Why do they coincide for a single decision?
4. code **Soft backup on the lake.** The proof in Section 14.2.2.3 used one property of the maximum, $|\max_a f(a) - \max_a g(a)| \le \max_a |f(a) - g(a)|$.
   1. Let $m_\beta(f) = \beta \log \sum_a \pi_{\text{ref}}(a)\, e^{f(a)/\beta}$. Show that m_β is monotone and that $m_\beta(f + c) = m_\beta(f) + c$ for every constant c, and deduce that m_β has the same property as the maximum. Conclude that the backup Equation 15.30 is a γ-contraction for every $\beta > 0$ and every reference.
   2. Apply `soft_v` to the slippery lake of Section 14.2, built as a `d2l.TabularMDP`, with a uniform reference. Plot $V_\beta(s_0)$ against β on a logarithmic grid from 10^{-4} to 10^2. Which two values computed in Section 14.2 does the curve approach at its two ends? Explain with the regularized objective why the curve is monotone.
   3. At about which β does $V_\beta(s_0)$ lie halfway between the two limits? Compare this coefficient with the differences between the optimal action values $Q^*(s_0, a)$ of the lake.
5. [extended] **Neural reward models.** ▣ The linear fit of `r_hat` leaves the pairs without data near zero, whereas a neural reward model extrapolates according to its inductive bias. Replace the 64 weights `w` by the outputs of a multilayer perceptron with two hidden layers of 64 ReLU units, one output per state-action pair, and set the row of the goal to zero. Train it on the same `D` and `y` with the Bradley-Terry negative log-likelihood, Adam at learning rate 0.01, for 2000 full-batch steps.
   1. Train one model whose input is a one-hot code of the state and the action, and one whose input is the two grid coordinates of the state, scaled to $[0, 1]$, together with a one-hot code of the action, with four seeds each. For every model, report the rms error on `core`, the fitted rewards of the moves `into` the hazard cells, and the true return of `plan`.
   2. Repeat the sweep of Section 15.3.4.3 on each fitted reward. Which representation needs the penalty? Which property of its input explains the difference?
6. code **Exploited reward errors.** ▣ Each run below modifies $\hat{r}$ and repeats the sweep of Section 15.3.4.3 with the same reference π_{ref}.
   1. Add $\delta \in \{0.25, 0.5, 1, 2\}$ to `r_hat[9, 3]`, the move up from state 9 into hazard cell 5, which never occurs in the data. How do the height and the divergence of the peak of the true return move with δ?
   2. Subtract 2 from `r_hat[9, 3]` instead. In a second run, subtract 2 from `r_hat[1, 1]`, a move into the hazard lane that $\hat{r}$ prices near zero. Before running them, predict how each change moves the peak of the true return. Explain both outcomes.
   3. Refit $\hat{r}$ from 1000 trajectories of the uniform policy, and keep π_{ref} as the reference of the penalty. Which error of the new fit does `plan` exploit? Does the true return still peak at an intermediate divergence?
7. **Slope of the frontier.** ▣ Write $u = 1/\beta$, and let $R(u)$ and $D(u)$ denote the expected reward and the divergence $D_{\text{KL}}(\pi^\star \| \pi_{\text{ref}})$ of the optimum Equation 15.27.

1. Show that $R(u) = \frac{d}{du}\log Z$ and that $R'(u) = \mathrm{Var}_{\pi^\star}(r)$. Then use the value $\beta \log Z$ of the proposition to show that $D'(u) = u\,\mathrm{Var}_{\pi^\star}(r)$.
2. Conclude that the frontier traced by `front` and `kl_f` has slope β at the point $\pi^\star$, and that it is increasing and concave whenever r is not constant on the support of π_{ref}.
3. Show that $\pi^\star$ also solves the constrained problem $\max_\pi \mathbb{E}_\pi[r]$ subject to $D_{\text{KL}}(\pi\|\pi_{\text{ref}}) \le D(u)$. Which quantity of the constrained problem does β become?
4. Best-of-N sampling draws N independent samples from π_{ref} and keeps the one with the highest reward. When the reward has a continuous distribution under the reference, show that the divergence of the best-of-N distribution from the reference is $\log N - (N-1)/N$. For the five-action bandit `p_ref`, `r`, whose rewards are distinct and increasing, best-of-N picks action a with probability $F(a)^N - F(a-1)^N$, where $F(a)$ is the reference probability of the actions up to a. Plot its expected reward against its divergence for $N = 1, \dots, 16$, together with `front` and `kl_f`. How close does it come to the frontier, and does its divergence stay below $\log N - (N-1)/N$?

8. **A moving reference.** ⊠ The trust region of Section 15.2 measures divergence from the previous iterate. This section showed that the idealized update $\pi_{k+1} = \mathrm{argmax}_\pi\ \mathbb{E}_\pi[r] - \tilde\beta\, D_{\text{KL}}(\pi\|\pi_k)$, started at $\pi_0 = \pi_{\text{ref}}$, converges to the argmax.
   1. Now add the fixed-reference penalty of Equation 15.26, as PPO-style fine-tuning does, so that π_{k+1} maximizes
   $$\mathbb{E}_\pi[r] - \beta\, D_{\text{KL}}(\pi\|\pi_{\text{ref}}) - \tilde\beta\, D_{\text{KL}}(\pi\|\pi_k). \tag{15.31}$$
   Show that, up to a constant,
   $$\log \pi_{k+1} = \frac{\beta \log \pi_{\text{ref}} + \tilde\beta \log \pi_k + r}{\beta + \tilde\beta}, \tag{15.32}$$
   and find the limit of π_k. Which coefficient determines the limit? How fast is the limit approached?
   2. Show that the recursion of part a is the update of `solve_kl` with a suitable step size, and express that step size through β and $\tilde\beta$. Which of the convergent step sizes of `solve_kl` have no interpretation as a trust region?
9. code **Forward-KL penalty.** ⊠ Section 15.3.3.5 contrasts the reverse divergence of Equation 15.26 with the forward one. Replace the penalty by the forward divergence,
   $$\max_\pi\ \mathbb{E}_{a\sim\pi}\big[r(a)\big] - \beta\, D_{\text{KL}}(\pi_{\text{ref}}\|\pi). \tag{15.33}$$
   1. Introduce a multiplier m for the constraint $\sum_a \pi(a) = 1$, and show that the optimum is $\pi(a) = \beta\,\pi_{\text{ref}}(a)/(m - r(a))$, where $m > \max_a r(a)$ is the unique value for which these probabilities sum to one.
   2. Let $\Delta(a) = \max_{a'} r(a') - r(a) > 0$ be the reward gap of a suboptimal action a. As $\beta \to 0$, how fast does $\pi(a)/\pi_{\text{ref}}(a)$ vanish under the forward penalty, and how fast under the reverse penalty?
   3. For the five-action bandit `p_ref`, `r`, compute the forward optimum by bisection on m. For each penalty, find the β at which the optimum's expected reward is 2.9, and compare the two optima action by action. Which actions receive more probability under the forward penalty, and how does this match Section 15.3.3.5?

10. code **Error modulo shaping.** ▯▯ Potential-based shaping (Section 14.2, (Ng et al. 1999)) adds

$$F(s,a) = \gamma \sum_{s'} P(s' \mid s,a)\, \Phi(s') - \Phi(s) \tag{15.34}$$

to the reward, for a potential Φ on the states with $\Phi(15) = 0$ at the absorbing goal.

1. Show that if Q_β solves the soft backup Equation 15.30 for the reward r, then $Q_\beta(s,a) - \Phi(s)$ solves it for $r + F$. Conclude that, for every β, the shaped reward has the same optimum π_β with the same divergence from π_{ref}, and that every policy's return from state 0 changes by $-\Phi(0)$.
2. Show that F changes the discounted return $\mathbf{c}(\tau)^\top r$ of every trajectory that reaches the goal by exactly $-\Phi(0)$. Which trajectories in `X` carry information about F, and through which factor?
3. The pairs of `core` neither start in nor enter the hazard cells 5 and 6, so they do not determine $\Phi(5)$ and $\Phi(6)$. On which pairs of `core` is the shaping term `F @ Phi` computed in this section largest? Why does the weight decay of the logistic regression push the fit in this direction?
4. Christiano et al. (2017) compare short trajectory segments that start at different states. Comparisons of this kind identify the reward up to transitions that the dynamics rule out (Skalse et al. 2023). Which step of part b fails for such segments?

15.4 Deep Q-Networks

In Section 14.3 an agent learned a table of action values one transition at a time, from transitions that any behavior could have produced. In Section 14.8 we replaced a policy table by a network and trained it on CartPole. Doing the same for Q-learning looks like a one-line change: let a network output the two action values and keep the update. For the policy gradient the swap cost nothing, because nothing in its derivation used the table. For Q-learning the table sits inside the target, $r + \gamma \max_{a'} Q(s',a')$. Once the table is a network, every update moves the target together with the prediction, one update changes the values of states it never visited, and the contraction argument that made the tabular method converge no longer applies. Can a network keep the table's licence to learn from any transition?

In 2013 a group at DeepMind made the swap work: one network, with one set of hyperparameters, learned to play dozens of Atari games from the screen pixels, many of them about as well as a human tester (Mnih et al. 2013, 2015). They called it a Deep Q-Network (DQN). Two devices kept it stable, a store of past transitions to train on and a frozen copy of the network to compute targets from. Both have been part of almost every off-policy deep method since. They also give the agent what the preceding sections could not: actor-critic used each batch once and PPO for a few epochs, while DQN trains on every transition it has stored, many times over.

We first show why the naive swap can fail, on a seven-state example whose weights grow without bound although there is no sampling noise. We then build DQN on CartPole, remove the frozen copy to see what it prevents, and end with the bias that the maximum in the target creates and a small change that reduces it. Section 15.3 replaced that maximum by a soft one; here it stays hard.

```
import copy
import gymnasium as gym
```

```
torch.set_num_threads(1)
```

We again use CartPole with discount 0.99, but specify the training budget as $50{,}000$ environment steps because episode length changes as the policy improves. The agent takes one gradient step after every two environment steps, on a minibatch of 128 stored transitions. Its Q-network maps a state through two hidden layers of 128 units to one value per action, and the greedy action is the argmax. Exploration is ϵ-greedy, with ϵ falling linearly from 1 to 0.05 over the first 10,000 steps, the `linear_schedule` of Section 14.3. `make_env` builds the environment and `make_qnet(seed)` a seeded Q-network. Their argument `copies` offers each action several times; it serves the last experiment of the section and is 1 until then.

```
gamma, num_env_steps, num_seeds = 0.99, 50_000, 3
buffer_size, batch_size, lr = 200_000, 128, 1e-3
train_freq, sync_every, warmup, grad_clip = 2, 250, 1_000, 10.0
epsilon = d2l.linear_schedule(1.0, 0.05, 10_000)

def make_env(copies=1):
    """CartPole; with copies > 1 each of its two actions is offered `copies`
    times (action a moves the cart as action a % 2), which leaves Q* as is."""
    env = gym.make('CartPole-v1')
    return env if copies == 1 else gym.wrappers.TransformAction(
        env, lambda a: a % 2, gym.spaces.Discrete(2 * copies))
```

```
def make_qnet(seed, copies=1):
    """A seeded Q-network with one output per action."""
    torch.manual_seed(seed)
    return nn.Sequential(nn.Linear(4, 128), nn.ReLU(),
                         nn.Linear(128, 128), nn.ReLU(),
                         nn.Linear(128, 2 * copies))
```

15.4.1 Sources of Instability

15.4.1.1 The Semi-Gradient Update

Replace the table of Section 14.3 by a network Q_w with parameters w, and the update stays what it was, a semi-gradient step toward the bootstrapped target. Sample a transition (s, a, r, s'), form the target

$$y = r + \gamma \left(1 - \mathbf{1}(s' \text{ terminal})\right) \max_{a'} Q_w(s', a'), \tag{15.35}$$

hold y fixed, and take one optimizer step that moves $Q_w(s, a)$ toward y. The indicator removes the bootstrap when s' ends the episode. Each step is a step of ordinary regression toward a fixed number. The sequence of steps is not descent on any fixed objective, though, because the number moves with w, which Section 14.8 observed for every bootstrapped target. A falling loss therefore shows that the network fits its current targets, not that the targets are right.

For a table, the expected update is a damped step of value iteration, and the Bellman backup is a contraction (Section 14.2): it shrinks the largest error over all states by the factor γ. A

network cannot represent every backed-up function, so each regression returns the function in its class closest to the targets. "Closest" is measured by least squares over the training data, in which the error at a state counts in proportion to how often the data visit it; for linear features this fit is a projection. The backup shrinks the largest error, while the fit shrinks a visitation-weighted average error. The two measures disagree, and a fit that reduces the weighted error can enlarge the error at a rarely visited state, which the next backup carries into the targets of the states that lead there. When the data come from the policy being evaluated, the two measures agree well enough for linear features to converge (Tsitsiklis and Van Roy 1997). With data from another policy, even linear features can diverge.

15.4.1.2 Correlated Data and Moving Targets

Two couplings make the network case harder in practice than the table case. The first is in the data. Consecutive transitions of one episode are nearly identical, since the cart and pole move only slightly from one step to the next, so training on the stream as it arrives updates a narrow region of the state space many times in a row. A table changes only the visited entry, so correlation only slows it down. A network changes its predictions at many states with every step (Figure 14.16), so a run of similar updates can overwrite what it learned elsewhere.

The second coupling is in the targets. The target y contains Q_w, so every optimizer step changes both the prediction and the number it is regressed toward. The critic of Section 15.1 had the same coupling, but it was refitted on fresh transitions of the current policy after every update, so an error at the states the policy visits was soon exposed and corrected.

15.4.1.3 The Deadly Triad and Baird's Counterexample

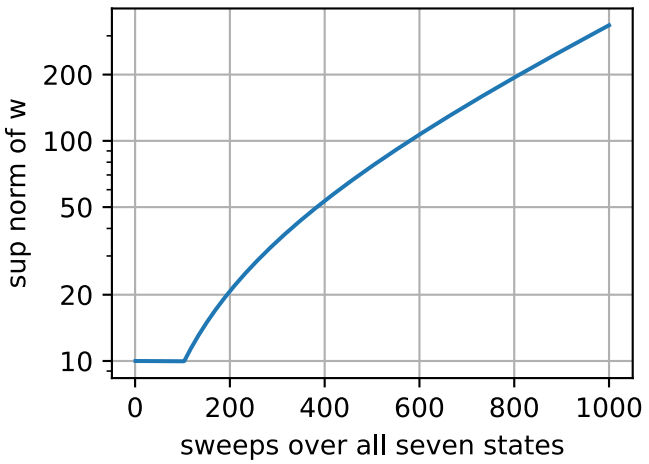
Three ingredients meet in the network version of Q-learning: *function approximation*, *bootstrapping*, and *off-policy data*, that is, data collected by a policy other than the one whose values are learned. Figure 15.8 places the algorithms of this chapter and the last by the ingredients they use. Each region with two ingredients holds a method that converges under its own conditions. Tabular Q-learning bootstraps from off-policy data and converges under the step-size conditions of Section 14.3. Linear temporal-difference learning, the critic of actor-critic in its simplest form, bootstraps with function approximation and converges for linear features when the data come from the evaluated policy. Behavior cloning fits a network to off-policy data without bootstrapping, which is supervised regression to fixed labels. These are sufficient conditions, not a theorem about pairs: nonlinear temporal-difference learning can diverge even on-policy (Tsitsiklis and Van Roy 1997). The name *deadly triad* refers to all three ingredients at once (Sutton and Barto 2018; Hasselt et al. 2018).

Baird's counterexample isolates the triad in a seven-state linear problem (Baird 1995). Every reward is zero, so the true value function is zero and is represented exactly by $w = 0$. The computation uses expected temporal-difference updates, so there is no sampling noise. It weights the seven states uniformly, an off-policy distribution, while it evaluates a policy that always moves to state 7:

```
w = np.ones(8)                          # eight weights for seven states
w[6] = 10.0                             # the classic starting point
Phi = np.zeros((7, 8))                  # linear values v = Phi w
Phi[:6, :6], Phi[:6, 7] = 2 * np.eye(6), 1.0   # v(i) = 2 w_i + w_8
Phi[6, 6], Phi[6, 7] = 1.0, 2.0                # v(7) = w_7 + 2 w_8
```

```
sup = []
for _ in range(1000):
    v = Phi @ w                              # every reward is 0
    delta = gamma * v[6] - v                 # the next state is always state 7
    w += 0.01 * (delta[:, None] * Phi).mean(0)   # uniform state weighting
    sup.append(np.abs(w).max())
print(f'sup norm of w after 0, 500, 1000 sweeps: 10, {sup[499]:.0f}, '
      f'{sup[-1]:.0f}')
print(f'the value the weights claim for state 7: {(Phi @ w)[6]:.0f}, '
      f'true value 0')
d2l.plot(np.arange(1, 1001), sup, 'sweeps over all seven states',
         'sup norm of w', yscale='log')
```

```
sup norm of w after 0, 500, 1000 sweeps: 10, 77, 335
the value the weights claim for state 7: 677, true value 0
```



After a transient, the weights grow by a factor per sweep that settles to a constant: after a thousand sweeps their sup norm has grown from 10 to 335, and the value that the weights claim for state 7 is 677, although its true value is zero. No sampling error is involved, and a smaller step size slows the divergence but does not remove it. Since the true value function is representable, the cause is not an approximation error either. Under the off-policy weighting, the expected update has a direction along which the weights grow (Sutton and Barto 2018). Weighting the states by the evaluated policy's own visits instead stops the divergence, although the values it reaches are wrong at the states that this weighting never visits (exercise 1).

Replay and target networks do not remove an ingredient of the triad. They reduce temporal correlation and slow the changes of the targets, which makes training much more stable in practice but does not restore a general convergence guarantee.

15.4.2 Replay and Target Networks

DQN answers the two couplings with two devices, one for each. Figure 15.9 shows how they fit together.

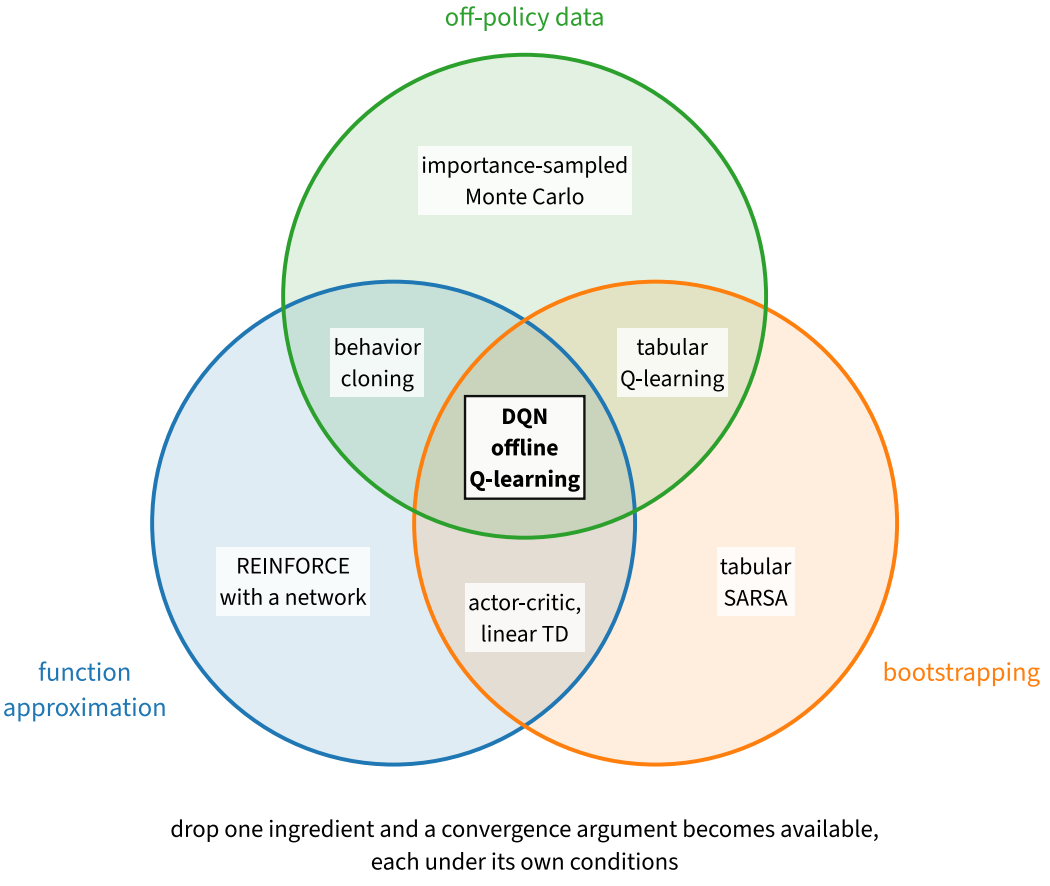


Figure 15.8: The deadly triad. Each circle is one ingredient, and every region is occupied by an algorithm these two chapters have taught; the center, all three ingredients at once, is where DQN and offline Q-learning live, and where Baird's counterexample shows that divergence is possible. Drop any one ingredient and a convergence argument becomes available, each under its own step-size, coverage and representation conditions: the triad is a warning pattern, not an if-and-only-if theorem.

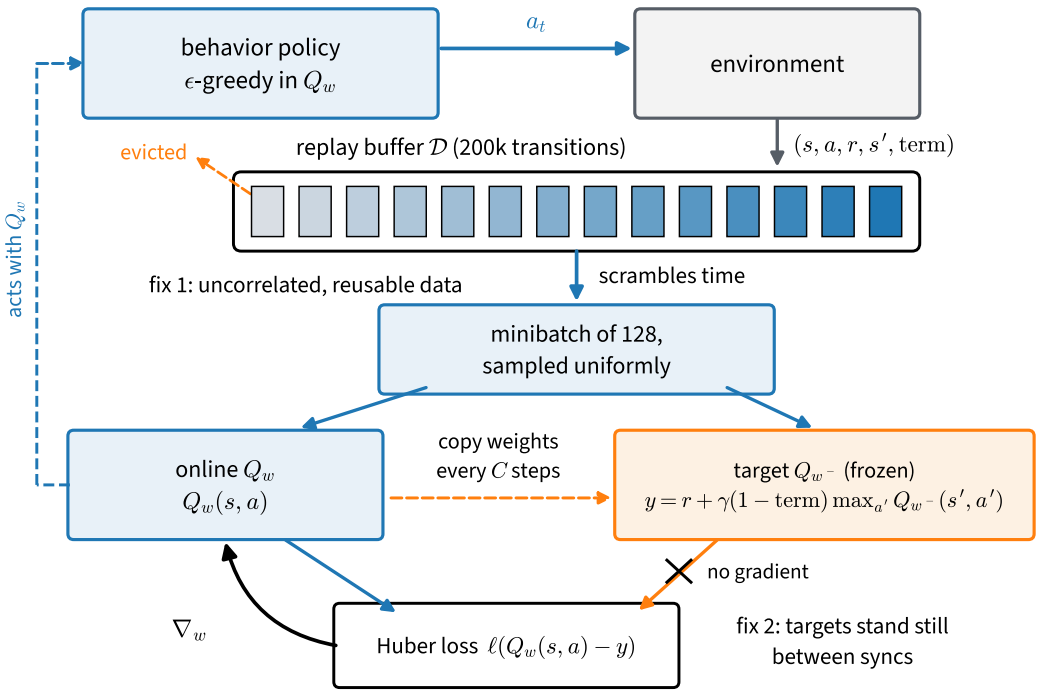


Figure 15.9: Deep Q-learning as a data flow. The behavior policy acts with the online network; transitions enter the replay buffer, and uniformly sampled minibatches scramble time, which makes the data nearly independent and reusable. The regression target is computed by a frozen copy of the network, refreshed every C steps, so the targets stand still between syncs. The gradient flows along exactly one path, into the online network; the marked edge from the frozen copy carries no gradient. Each stored transition carries the flag term, and the mask $1 - \text{term}$ in the target stops the bootstrap only at a true termination. The drawn buffer is full and evicting its oldest entries, the steady state of a long run; in this section's runs the 50,000-step budget never fills the 200,000-slot ring, so nothing is evicted.

15.4.2.1 Experience Replay

The remedy for correlated data is an *experience replay buffer* (Lin 1992). Every transition is stored in a large array, and each training minibatch is drawn uniformly from the array. A minibatch then combines transitions collected at different times, and each transition is used in many updates. Q-learning permits this because the target for (s, a, r, s') does not depend on the policy that chose a. Replay has a drawback: most stored transitions were collected by older policies, so an error in the targets can persist over many updates until the current behavior revisits the states where it lives.

`ReplayBuffer` keeps the five fields of a transition in preallocated arrays: the state, the action, the reward, the next state, and the flag `term` that records termination. `add` writes a transition at the next free slot, overwriting the oldest one once the array is full, and `sample` draws `batch_size` slots uniformly and returns them as the `Batch` of Section 14.6:

```
class ReplayBuffer:
    """A ring of transitions in preallocated numpy; sample() returns a Batch."""
    def __init__(self, capacity, obs_dim):
        self.obs = np.zeros((capacity, obs_dim), np.float32)
        self.act = np.zeros(capacity, np.int64)
        self.rew = np.zeros(capacity, np.float32)
        self.next_obs = np.zeros((capacity, obs_dim), np.float32)
        self.term = np.zeros(capacity, np.float32)
        self.capacity, self.size, self.ptr = capacity, 0, 0

    def add(self, obs, act, rew, next_obs, term):
        i = self.ptr
        self.obs[i], self.act[i], self.rew[i] = obs, act, rew
        self.next_obs[i], self.term[i] = next_obs, term
        self.ptr, self.size = (i + 1) % self.capacity, min(self.size + 1,
                                                           self.capacity)

    def __len__(self):
        return self.size

    def sample(self, batch_size, rng):
        """batch_size transitions drawn uniformly, each a one-step
        fragment: ep_ends marks every transition as its own episode."""
        i = rng.integers(self.size, size=batch_size)
        return d2l.Batch(self.obs[i], self.act[i], self.rew[i],
                         self.next_obs[i], self.term[i],
                         np.arange(1, batch_size + 1))
```

The capacity in this section is $200{,}000$ transitions, larger than the $50{,}000$-step budget, so no transition is evicted; exercise 5 studies a smaller buffer. `sample` declares every transition an episode fragment of length one, because a uniform draw has no episode order. On such a batch `reward_to_go` is the reward itself, and the `truncation_bootstrap` of Section 14.8 adds $\gamma(1-\text{term})\,\widehat{V}(s')$ for a given value estimate $\widehat{V}$, so their sum is the one-step target that `td_target` from Section 15.1 computes. Replay leaves only the one-step target, which is why it pairs with one-step bootstrapping. An n-step target needs consecutive transitions, so the buffer would have to keep their order (exercise 7).

15.4.2.2 The Target Network

The remedy for moving targets is the *target network* (Mnih et al. 2015). A frozen copy Q_{w^-} of the online network computes every bootstrap,

$$y = r + \gamma \left(1 - \mathbf{1}(s' \text{ terminal})\right) \max_{a'} Q_{w^-}(s', a'), \tag{15.36}$$

and the copy is synchronized with the online weights, $w^- \leftarrow w$, every C environment steps, here 250. Between synchronizations the targets stand still, and the online network solves an ordinary regression problem. Predictions feed back into targets once per synchronization instead of once per gradient step. The critic of Section 15.1 bootstrapped from its newest weights, because each batch was fresh; replay removes that freshness, and DQN accepts targets a few hundred steps old in exchange for a regression problem that changes slowly.

Two further controls bound the size of a step. The Huber loss is quadratic for residuals below 1 and linear above, so its derivative with respect to the prediction never exceeds 1. The full gradient also contains $\nabla_w Q_w(s, a)$, which the Huber loss does not bound, so the update clips the norm of the parameter gradient at 10, the default of Stable-Baselines3 rather than of the original paper (Raffin et al. 2021). In code, `q_values` runs the network on a NumPy state without recording gradients, `fit_q` takes one clipped Huber step of $Q_w(s, a)$ toward fixed targets y, and `q_step` is the DQN update: the targets are `batch.td_target` with the bootstrap $\max_{a'} Q_{w^-}(s', a')$ of the frozen copy.

```
def q_values(qnet, obs):
    """The acting forward: numpy in, numpy out, no gradient recorded."""
    with torch.inference_mode():
        return qnet(torch.as_tensor(obs)).numpy()

def fit_q(qnet, opt, batch, y):
    """One clipped Huber regression step of Q_w(s, a) toward the fixed y."""
    pred = qnet(torch.as_tensor(batch.obs)).gather(
        -1, torch.as_tensor(batch.act)[:, None]).squeeze(-1)
    loss = nn.functional.huber_loss(pred, torch.as_tensor(y))
    opt.zero_grad()
    loss.backward()
    nn.utils.clip_grad_norm_(qnet.parameters(), grad_clip)
    opt.step()

def q_step(qnet, target, opt, batch):
    """The DQN update: the critic's own td_target, bootstrapped by the copy."""
    fit_q(qnet, opt, batch, batch.td_target(
        lambda s: q_values(target, s).max(-1), gamma))
```

15.4.2.3 The Training Loop

The parts of the learner that differ between frameworks come from `make_learner`: the frozen copy, the optimizer, `update`, which takes one step of the update rule `step` on a sampled batch, and `sync`, which copies the online weights into the frozen copy.

```
def make_learner(qnet, step):
    """DQN's moving parts: the frozen copy, Adam, the update and the sync."""
    target = copy.deepcopy(qnet)
    opt = torch.optim.Adam(qnet.parameters(), lr=lr)
    update = lambda b: step(qnet, target, opt, b)
    sync = lambda: target.load_state_dict(qnet.state_dict())
    return update, sync
```

The loop acts ϵ-greedily with `epsilon_greedy` of Section 14.3, which takes a random action with probability ϵ and the argmax otherwise, stores every transition, and after the warm-up of 1,000 steps calls `update` on a sampled minibatch every second step. One switch sets up the experiment of this section. With `use_target=True` the copy is refreshed every `sync_every` steps. With `use_target=False` it is refreshed after every step, so the target is computed by the online network itself, the naive recipe without a target network. The argument `step` selects the update rule, and `copies` the number of copies of each action. At the end of every episode the generator yields the step count, the episode's return, and the network's estimate $\max_a Q_w(s_0, a)$ at the centered start state $s_0 = \mathbf{0}$.

```
def train_dqn(seed, qnet, use_target=True, step=q_step, copies=1):
    """DQN on CartPole; yields (env step, episode return, max_a Q(s0, a))."""
    rng, env = np.random.default_rng(seed), make_env(copies)
    update, sync = make_learner(qnet, step)
    buffer, s0 = ReplayBuffer(buffer_size, 4), np.zeros(4, np.float32)
    obs, ep_return = env.reset(seed=seed)[0], 0.0
    for t in range(1, num_env_steps + 1):
        a = d2l.epsilon_greedy(q_values(qnet, obs), epsilon(t), rng)
        next_obs, rew, terminated, truncated, _ = env.step(a)
        buffer.add(obs, a, rew, next_obs, float(terminated))
        obs, ep_return = next_obs, ep_return + rew
        if terminated or truncated:
            yield t, ep_return, q_values(qnet, s0).max()
            obs, ep_return = env.reset()[0], 0.0
        if len(buffer) >= warmup and t % train_freq == 0:
            update(buffer.sample(batch_size, rng))
        if t % (sync_every if use_target else 1) == 0:
            sync()
```

The buffer stores `terminated`, never `truncated`. A CartPole episode cut off at 500 steps ends in a state that has a future, and Section 14.3 showed that masking the bootstrap there teaches the network that surviving is worthless; exercise 3 traces what would go wrong.

15.4.2.4 The Target-Network Ablation

Is the frozen copy needed? We train three seeds of each arm, identical except for the switch. Statements below about how often something happens across seeds were checked over twenty seeds per arm. The networks are built outside the training loop so that later cells can evaluate the trained agents. `train_checked` wraps `train_dqn`. It collects the per-episode records and, at the first episode end past every multiple of 5,000 environment steps, runs the *greedy policy*, which always takes $\operatorname{argmax}_a Q_w(s, a)$, for ten episodes on a separate copy of the environment, and once more at the end of the budget; `d2l.evaluate` is the Monte Carlo evaluation of Section 14.2, which averages the returns of a policy over a number of episodes,

and `greedy` turns a Q-network into such a policy. These checks do not change training; we read them once the value estimates are understood.

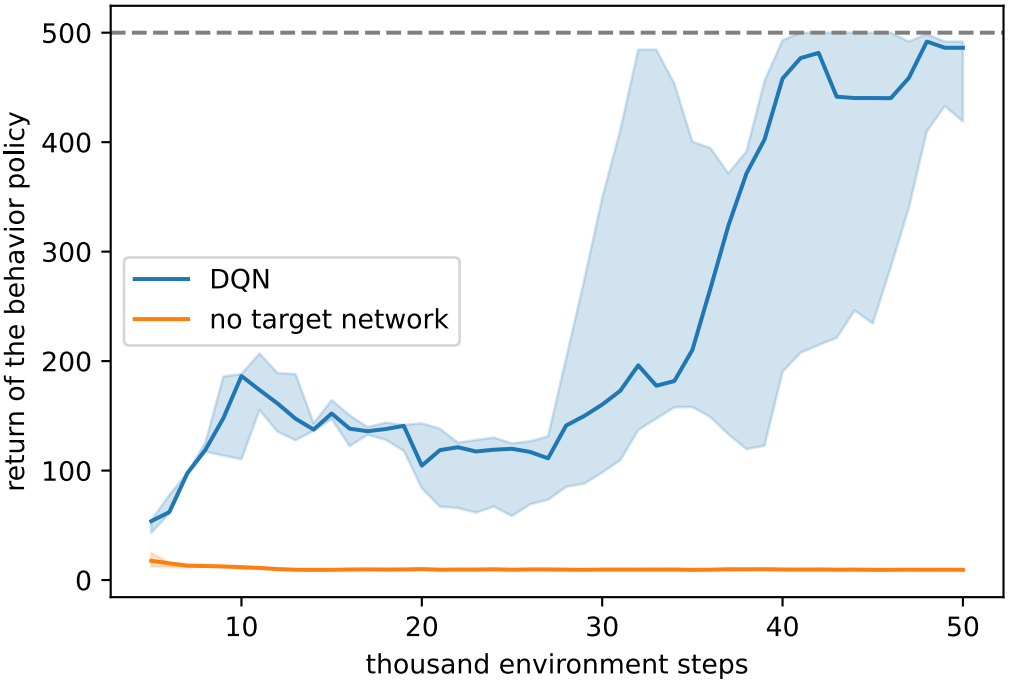
```
def greedy(qnet):
    """The greedy policy of a Q-network, (obs, rng) -> argmax_a Q(obs, a)."""
    return lambda s, rng: int(q_values(qnet, s).argmax())

def train_checked(seed, qnet, every=5_000, episodes=10, **kwargs):
    """train_dqn, plus a greedy evaluation over a few episodes at the first
    episode end past every multiple of `every` steps, and one at the end."""
    env, run, checks = make_env(kwargs.get('copies', 1)), [], []
    env.reset(seed=10_000 + seed)
    for t, ret, v0 in train_dqn(seed, qnet, **kwargs):
        run.append((t, ret, v0))
        if t >= every * (len(checks) + 1):
            checks.append(d2l.evaluate(env, greedy(qnet), episodes))
    checks.append(d2l.evaluate(env, greedy(qnet), episodes))
    return run, checks
```

```
arms = ('DQN', 'no target network')
qnets = {arm: [make_qnet(seed) for seed in range(num_seeds)] for arm in arms}
runs, checks = {}, {}
for arm in arms:
    out = [train_checked(seed, qnets[arm][seed], use_target=arm == 'DQN')
           for seed in range(num_seeds)]
    runs[arm], checks[arm] = [r for r, _ in out], [c for _, c in out]
```

Seeds finish different numbers of episodes, so we plot against environment steps, the unit the budget is written in. `d2l.on_grid` resamples each run's per-episode records onto a common grid of environment steps, and `d2l.plot_curves` draws the median over seeds with a band from the lowest to the highest seed:

```
grid = np.arange(1, num_env_steps // 1000 + 1) * 1000
d2l.plot_curves({arm: d2l.on_grid(runs[arm], grid, 1) for arm in runs},
                x=grid / 1000, xlabel='thousand environment steps',
                ylabel='return of the behavior policy', smooth=5,
                reference=500)
```

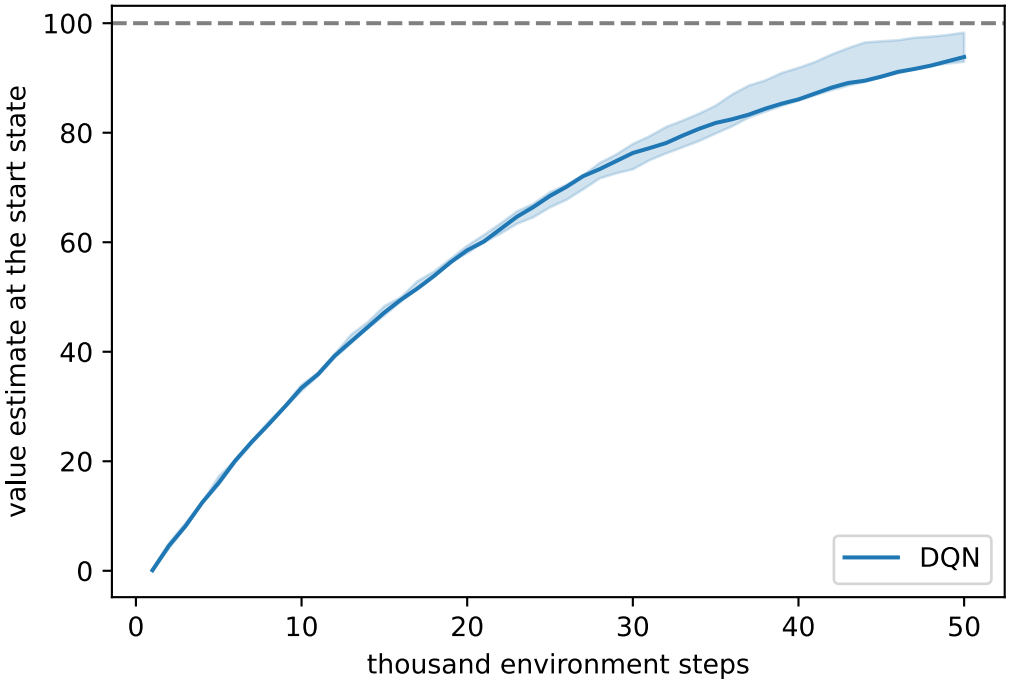


With a target network every seed improves from the random policy's return of about 20 to returns in the hundreds, although the curves stay ragged. The behavior return understates the policy, because the final exploration rate $\epsilon = 0.05$ still forces a random action on one step in twenty. Without a target network the return never rises above the random policy's and soon falls to about 9, the length of an episode in which the cart is pushed the same way at every step.

The value estimates show what went wrong. The training loop records $\max_a Q_w(s_0, a)$ at the centered start state, and we can say in advance where it should end. The buffer masks the bootstrap only on `terminated`, so the update keeps bootstrapping across the 500-step limit, and the state contains no clock. The network therefore learns the value of a continuing task, in which the time limit stops data collection but not the process. A policy that balances the pole forever earns $\sum_t \gamma^t = 1/(1-\gamma) = 100$ from the start, and no policy earns more, because every step pays 1. The dashed line marks this ceiling:

```
ceiling = 1 / (1 - gamma)
d2l.plot_curves({'DQN': d2l.on_grid(runs['DQN'], grid, 2)},
                x=grid / 1000, xlabel='thousand environment steps',
                ylabel='value estimate at the start state',
                reference=ceiling)
print(f'continuing-task ceiling: {ceiling:.0f}; the no-target arm ends at ' +
      ', '.join(f'{np.asarray(r)[-1, 2]:.0e}' for r in
                runs['no target network']))
```

```
continuing-task ceiling: 100; the no-target arm ends at 1e+08, 1e+08, 1e+08
```



DQN's estimate climbs toward the ceiling, while the estimate without a target network passes 10^8. The runaway goes up rather than in an arbitrary direction because CartPole pays 1 at every step. Every target of a transition that does not end the episode is $1 + \gamma \max_{a'} Q_w(s', a')$, one more than a discounted output of the network itself, and the shared parameters that lift $Q_w(s, a)$ toward it also lift $Q_w(s', \cdot)$, because s' is close to s. The target stays ahead of the prediction at every step, and nothing stops the climb.

A frozen copy holds the target still for 125 gradient steps, so values can climb only one synchronization at a time. Suppose every value starts at zero and the online network fits its targets exactly between two synchronizations. Then after n synchronizations no value exceeds

$$\sum_{k=0}^{n-1} \gamma^k = \frac{1 - \gamma^n}{1 - \gamma},$$

the discounted return of n steps of reward 1. The budget holds about 196 synchronizations after the first gradient step, which bounds the start-state value by about 86. The plotted estimate follows this bound closely, a few points above it from about 10,000 steps on, and ends above 86 on every seed, because a network generalizes: a step that lifts the value of one state lifts its neighbors too, so value spreads faster than one backup per synchronization (exercise 6).

A target network is one repair for this feedback, not the only one. PQN (Gallici et al. 2025) drops both the frozen copy and the replay buffer, normalizes the activations of every hidden layer, and collects data from many environments in parallel; exercise 10 tries the normalization on CartPole.

15.4.2.5 Evaluation of the Greedy Policy

The return curve is a poor summary of a run. It includes the exploratory actions, its final value depends on when the run happens to stop, and its best value is chosen after looking at the whole curve, which flatters every method. We therefore evaluate the greedy policy at budgets fixed in advance. These are the checks that `train_checked` recorded, a ten-episode greedy return after every 5,000 environment steps and one at the end of the budget:

```
print('greedy return (10 episodes) every 5,000 environment steps')
for arm in arms:
    for seed, c in enumerate(checks[arm]):
        print(f'{arm:>17}, seed {seed}: ' + ' '.join(f'{x:3.0f}' for x in c))
```

```
greedy return (10 episodes) every 5,000 environment steps
              DQN, seed 0: 214 152  57 111 120 136 500 500 500 500
              DQN, seed 1: 232 150 101  93  54 163 101 335 404 103
              DQN, seed 2: 182  92 124 117 117 112 265 500 500 500
no target network, seed 0:   9   9  10   9   9   9   9   9   9  10
no target network, seed 1:  10  10   9   9  10  10   9   9   9  10
no target network, seed 2:   9   9  10  10   9   9   9   9   9   9
```

Checked this way, DQN's greedy policy sits mostly at a return of about 50 to 250 for the first 30,000 steps, with an occasional early check at 500, and then reaches 500 on most seeds between 35,000 and 45,000 steps. A good check is not final: some seeds later fall back by hundreds of points, as one seed of the printout does, and about half of the twenty did. Without a target network the greedy policy never lasts more than about 10 steps at any check, so training longer does not rescue it. Although DQN trains on each stored transition many times, it needs more environment steps to balance the pole for the full episode than PPO with four epochs per batch needed in Section 15.2.

A greedy return says how good the policy is, not whether the network knows how good it is. Van Hasselt et al. (2016) test a value estimate against the discounted return that the greedy policy actually earns, which is the quantity the estimate claims to predict. `audit` computes three numbers for a trained network: the greedy return over 100 episodes, the greedy policy's discounted return over 20 episodes (`d2l.evaluate` with `gamma`), and the estimate $\max_a Q_w(s_0, a)$. Episodes start close to the centered state, so the discounted return measures the value of s_0. Under the 500-step limit even a perfect policy earns only $(1-\gamma^{500})/(1-\gamma) \approx 99.3$, slightly less than the ceiling of the continuing task.

```
def audit(qnet, seed, copies=1):
    """Greedy return over 100 episodes, its discounted return over 20, and
    the network's own estimate max_a Q(s0, a) of the latter."""
    env = make_env(copies)
    env.reset(seed=100 + seed)
    score = d2l.evaluate(env, greedy(qnet), 100)
    env.reset(seed=100 + seed)
    disc = d2l.evaluate(env, greedy(qnet), 20, gamma=gamma)
    return score, disc, q_values(qnet, np.zeros(4, np.float32)).max()

audits = {arm: [audit(qnets[arm][seed], seed) for seed in range(num_seeds)]
          for arm in arms}
for arm in arms:
```

```
    for seed, (score, disc, v0) in enumerate(audits[arm]):
        print(f'{arm:>17}, seed {seed}: return {score:3.0f}, discounted '
              f'{disc:5.1f}, max_a Q(s0) {v0:8.3g}')
```

```
              DQN, seed 0: return 500, discounted  99.3, max_a Q(s0)     93.9
              DQN, seed 1: return  93, discounted  61.9, max_a Q(s0)     98.4
              DQN, seed 2: return 500, discounted  99.3, max_a Q(s0)     93.1
no target network, seed 0: return  10, discounted   9.0, max_a Q(s0)  1.4e+08
no target network, seed 1: return   9, discounted   9.1, max_a Q(s0) 1.47e+08
no target network, seed 2: return   9, discounted   8.9, max_a Q(s0) 1.45e+08
```

Where the greedy policy balances the pole, the estimate sits within about eight points of the discounted return that the policy earns, usually below it, because the values are still climbing at the end of the budget: the plot of the start-state value above has not leveled off. Where the greedy policy fails, which happened on about two seeds in five, the estimate is above what the policy earns, by more than 20 points on about half of such runs. With two actions, these failing runs are the only sign of the network promising more than its policy delivers, too few and too mixed with other errors to measure the effect. A cleaner test needs a maximum over more actions.

15.4.3 Overestimation

15.4.3.1 The Maximization Bias in Numbers

The target Equation 15.36 maximizes estimated action values. For noisy estimates, $\mathbb{E}[\max_{a'} \hat{Q}(s', a')] \geq \max_{a'} \mathbb{E}[\hat{Q}(s', a')]$, with strict inequality when the noise can change which action is largest. Section 14.3 demonstrated this bias in a table, and Section 15.3 showed that the hard maximum is the $\beta \to 0$ limit of the soft backup with a uniform reference. With function approximation, the bias at one state feeds into the targets of the states that lead to it (Thrun and Schwartz 1993).

The following simulation gives four actions the same true value of zero and adds independent unit noise to each estimate. The single estimator selects and evaluates an action with the same noisy values. The double estimator selects with one set of noisy values and evaluates with an independent second set (Hasselt 2010):

```
rng = np.random.default_rng(0)
est = rng.standard_normal((100_000, 4))     # four values, all truly zero
second = rng.standard_normal((100_000, 4))  # an independent second estimate
sel = est.argmax(1)
print(f'single estimator, E[max of the estimates]: {est.max(1).mean():.3f}')
print(f'select with one, evaluate with the other: '
      f'{second[np.arange(100_000), sel].mean():+.3f}')
```

```
single estimator, E[max of the estimates]: 1.031
select with one, evaluate with the other: -0.006
```

The maximum of four unit-noise estimates has an upward bias of about one unit. With independent estimates for selection and evaluation, the evaluator's noise has mean zero whichever action was selected, and the measured bias is near zero. Figure 15.10 shows that the single estimator's bias grows with the number of actions, while the independent double estimator stays at zero. A language model choosing its next token among 10^5 actions

would face a far larger bias from a maximum over noisy estimates, one reason why language-model post-training uses policy gradients rather than a Q-network. A fixed dataset makes the bias worse still, because no new interaction corrects an overrated action (Section 15.7).

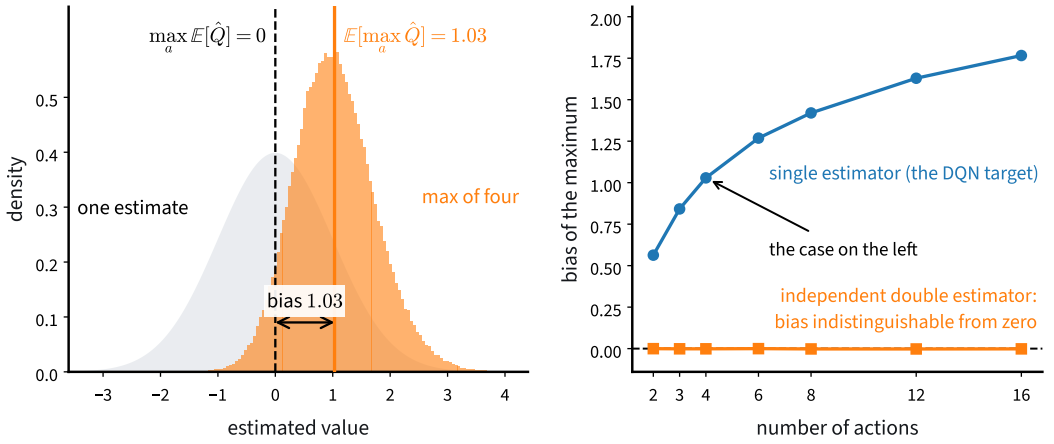


Figure 15.10: A maximum over noisy estimates is biased upward. Left: the distribution of the largest of four independent estimates of the same true value of zero, against the distribution of a single estimate; the maximum carries a measured bias of 1.03. Right: the bias of the single estimator grows with the number of actions, while the bias of the independent double estimator, one set of estimates selecting the action and an independent set scoring it, is indistinguishable from zero throughout.

15.4.3.2 Double DQN

The double estimator needs two value estimates, and DQN already has two networks. Double DQN (Van Hasselt et al. 2016) selects the action with the online network and evaluates it with the target network:

$$y = r + \gamma \left(1 - \mathbf{1}(s' \text{ terminal})\right) Q_{w^-}\left(s', \operatorname*{argmax}_{a'} Q_w(s', a')\right). \tag{15.37}$$

An action that the online network overrates by chance is then scored by a different network. The change needs no new network, only one more forward pass of the online network on s'. The two networks are not independent, since the copy is the online network of at most 125 gradient steps earlier, so Double DQN removes only part of the bias. If the evaluating estimate is correlated with the selecting one with coefficient ρ, the bias that remains is ρ times that of the single estimator (exercise 4). The change to `q_step` is three lines:

```
def q_step_double(qnet, target, opt, batch):
    """q_step with selection split from evaluation, eq_double_dqn."""
    sel = q_values(qnet, batch.next_obs).argmax(-1)
    fit_q(qnet, opt, batch, batch.td_target(
        lambda s: q_values(target, s)[np.arange(len(s)), sel], gamma))
```

15.4.3.3 Duplicated Actions

CartPole gives the maximum little to choose from. The bias of the larger of two independent unit-noise estimates is $1/\sqrt{\pi} \approx 0.56$, about half the four-action value, and the two-action estimates of the ablation are pulled the other way by the lag of the synchronizations. To see the bias on a trained network without changing the task, we offer each of CartPole's actions nine times. `make_env(copies=9)` exposes 18 actions, the size of the full Atari action set, and action a pushes the cart the way action $a \bmod 2$ does; `make_qnet(seed, copies=9)` has 18 outputs. The duplicates change neither the optimal values nor the ceiling of 100. The 18 outputs are nine noisy estimates of each of two values, and the maximum in the target picks the luckiest of nine, the right panel of Figure 15.10 on a real network. The next cell trains DQN and Double DQN with 18 actions, and Double DQN with two, and audits every trained network; DQN with two actions is the arm of the ablation:

```
arms2 = {'Double DQN, 2 actions': (q_step_double, 1),   # (update, copies)
         'DQN, 18 actions': (q_step, 9),
         'Double DQN, 18 actions': (q_step_double, 9)}
names = ('DQN, 2 actions', *arms2)
qnets[names[0]], runs[names[0]] = qnets['DQN'], runs['DQN']
audits[names[0]] = audits['DQN']
for name, (step, copies) in arms2.items():
    qnets[name] = [make_qnet(seed, copies) for seed in range(num_seeds)]
    runs[name] = [list(train_dqn(seed, qnets[name][seed], step=step,
                                 copies=copies)) for seed in range(num_seeds)]
    audits[name] = [audit(qnets[name][seed], seed, copies)
                    for seed in range(num_seeds)]
```

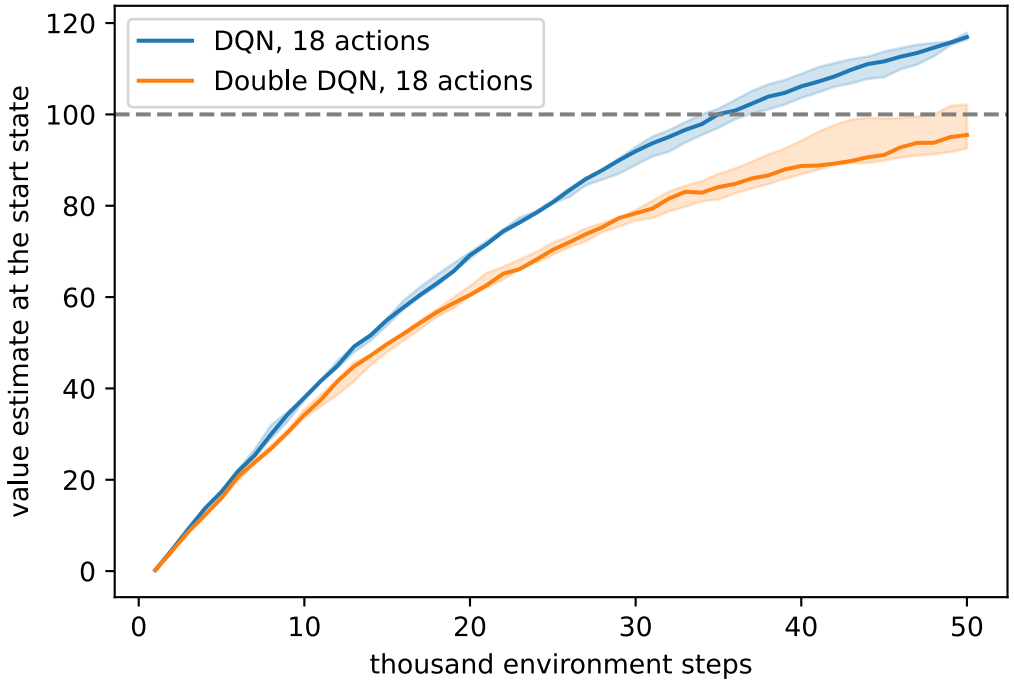
For each arm we print, per seed, the greedy return, the greedy policy's discounted return and the start-state estimate:

```
print(f'{"per seed:":>22} {"greedy return":>17}   {"discounted":>17}'
      f'   {"max_a Q(s0)":>17}')
for name in names:
    res = np.array(audits[name]).T
    print(f'{name:>22} ' + '   '.join(' '.join(f'{x:5.1f}' for x in row)
                                       for row in res))
```

```
            per seed:     greedy return          discounted         max_a Q(s0)
       DQN, 2 actions 500.0  93.4 500.0    99.3  61.9  99.3    93.9  98.4  93.1
 Double DQN, 2 actions 500.0 500.0 500.0    99.3  99.3  99.3    92.7  95.0  91.1
      DQN, 18 actions 181.0 500.0 153.2    83.8  99.3  78.7   117.7 116.2 117.1
Double DQN, 18 actions  94.0 356.2 500.0    61.4  95.8  99.3    95.5 102.0  92.4
```

The start-state estimates of the 18-action pair, against the ceiling, over training:

```
d2l.plot_curves({name: d2l.on_grid(runs[name], grid, 2) for name in names[2:]},
                x=grid / 1000, xlabel='thousand environment steps',
                ylabel='value estimate at the start state',
                reference=ceiling)
```



With each action offered nine times, DQN's start-state estimate ends above the ceiling on every seed, typically 10 to 30 points above it. Where its greedy policy balances the pole and earns about 99, the estimate exceeds that return by about as much. No value the task allows lies above the ceiling, so the overestimation is certain without any further evaluation. In the plot, DQN's estimate crosses the ceiling before 40,000 steps and keeps rising, while Double DQN's stays near or below it. Double DQN's estimate ends 10 to 25 points lower on almost every seed, near the ceiling, although a seed whose policy fails can still overshoot. A few of the twenty 18-action DQN runs are still climbing at 50,000 steps, the bias feeding back through the bootstrap. With two actions the two targets agree within a few points, and at this budget neither arm's greedy return differs beyond the variation across seeds. On CartPole the double estimator makes the values more accurate without changing how well the greedy policy balances the pole.

We can now train a value network on transitions from any policy, replayed many times, with a frozen copy that keeps the targets still and a second estimate that keeps the maximum from inflating them. Both repairs assume that the maximum over actions can be computed: CartPole's two pushes and Atari's 18 joystick actions form a list. The torque of a motor is a number in an interval, and an argmax over it is an optimization problem of its own at every step. The next section keeps replay and the frozen copy, and replaces the maximum by a second network trained to propose the best action, which completes the off-policy column of Figure 14.17.

15.4.4 Summary

A network in place of the Q-table combines function approximation, bootstrapping and off-policy data, and Baird's seven states show that this combination can diverge even when the true values are representable and no sampling noise is present. DQN keeps all three

ingredients and repairs two couplings: a replay buffer breaks the correlation of consecutive transitions and reuses each of them many times, and a frozen target network holds the regression targets still between synchronizations. On CartPole, removing the frozen copy sends the start-state value past 10^8; with it, the value climbs one synchronization at a time toward the ceiling of 100, and the greedy policy balances the pole for the full 500 steps on most seeds after 35,000 to 45,000 environment steps. The maximum in the target biases the values upward, more so the more actions it chooses among, and with each action offered nine times DQN's estimate ends above the ceiling on every seed. Double DQN selects the action with the online network and evaluates it with the frozen copy, which removes most of that bias at the cost of one extra forward pass.

15.4.5 Exercises

1. code **Baird under on-policy weighting.** The computation of Baird's counterexample in Section 15.4.1.3 weights the seven states equally through `.mean(0)`. In matrix form, one sweep is $\mathbf{w} \leftarrow \mathbf{w} + \alpha A \mathbf{w}$ with $\alpha = 0.01$ and

$$A = \Phi^\top D(\gamma P - I)\Phi,$$

   where D is the diagonal matrix of the state weights and P is the transition matrix of the evaluated policy, every row of which puts all probability on state 7.

   1. Compute the eigenvalues of A for $D = I/7$. Compare $1 + \alpha\lambda$, for the eigenvalue λ with the largest real part, with the per-sweep growth factor of the sup norm of $\mathbf{w}$ late in a run of $20{,}000$ sweeps.
   2. The evaluated policy moves to state 7 and stays there, so its visitation distribution puts all weight on state 7. Rerun the iteration with this weighting. To which weights and which state values does it converge, and how do these values compare with the true values?
   3. Mix the two weightings, $\mathbf{d} = (1-p)\,\mathbf{1}/7 + p\,\mathbf{e}_7$, and find by bisection the smallest p for which no eigenvalue of A has a positive real part, for $\gamma \in \{0.9, 0.95, 0.99\}$. How does the amount of off-policy weight that the iteration tolerates depend on the discount?

2. **Huber loss and norm clipping.** `fit_q` regresses with the Huber loss of threshold 1, $\ell(e) = e^2/2$ for $|e| \le 1$ and $\ell(e) = |e| - 1/2$ otherwise, where $e = Q_w(s,a) - y$ is the residual of one transition, and averages it over the B transitions of a minibatch. The update then rescales the minibatch gradient to norm `grad_clip` $= 10$ whenever its norm exceeds 10 and passes it to Adam.

   1. Show that $\ell'(e) = \max(-1, \min(1, e))$. In which sense is Huber regression squared-loss regression with clipped residuals?
   2. Show that the minibatch gradient has norm at most $\max_i \|\nabla_w Q_w(s_i, a_i)\|$, and explain why this bound can grow while every residual stays the same.
   3. Adam divides a running average of the gradients by the square root of a running average of their squares. Apart from the small constant it adds for numerical stability, multiplying every gradient of a run by one constant therefore leaves its steps unchanged. Which gradient sequences does the norm clip change? How does it change the Adam steps that follow a single gradient of very large norm, such as one that may follow a target-network synchronization?

4. An Adam step moves each parameter by roughly the learning rate, here 10^{-3}, whatever the size of its gradient. The run without a target network takes about $25{,}000$ such steps, and its start-state values reach about 10^8. Estimate how far one parameter can travel during the run, and explain how steps of that size can produce values of 10^8 in the three linear layers of `make_qnet`, whose hidden layers have 128 units.

3. **Terminated, not done.** `train_dqn` stores `float(terminated)`, so a transition into CartPole's 500-step time limit still bootstraps. Consider an implementation that stores `float(terminated or truncated)` instead.

   1. Which stored transitions does the change affect while the greedy policy still fails within a few hundred steps, and which once every episode reaches the limit?
   2. The observation contains no clock, so the network cannot tell a state at step 499 from a similar balanced state at step 50. Model its estimate on balanced states by a single number q, the value of a one-state task that pays reward 1 at every step and is cut off after H steps. With the faulty mask, $H-1$ of the H transitions of an episode have the target $1+\gamma q$ and the last one has the target 1. Show that the fixed point of the averaged update is $q = H/(H-\gamma(H-1))$, and evaluate it at $H = 500$ and $\gamma = 0.99$. The start-state value plot of Section 15.4.2.4 marks $1/(1-\gamma)$ as the ceiling. Compare q with the value of the start state of a 500-step episode when the remaining time is known, $(1-\gamma^H)/(1-\gamma)$. Which of the two is the ceiling of the faulty implementation for a network without a clock? What would a final estimate of 94 indicate?

4. code **Maximization bias in closed form.** Let the estimates $X_1, \dots, X_m$ of m actions with the same true value 0 be jointly normal with variance σ^2 each.

   1. For $m = 2$ and independent estimates, show that $\mathbb{E}[\max(X_1, X_2)] = \sigma/\sqrt{\pi}$, using $\max(x, y) = (x+y)/2 + |x-y|/2$.
   2. The outputs of one network share its hidden layers, so their errors are correlated. For $m = 2$ and correlation c between X_1 and X_2, show that the bias becomes $\sigma\sqrt{(1-c)/\pi}$.
   3. In Double DQN, the target network evaluates the action that the online network selects, and the errors of the two networks are correlated. Model the evaluating estimate of action a as $Y_a = \rho X_a + \sqrt{1-\rho^2}\, Z_a$, where $Z_1, \dots, Z_m$ have mean zero and variance σ^2 each and are independent of $X_1, \dots, X_m$, so that Y_a has variance σ^2 and correlation ρ with X_a. Show that the bias of $Y_{\text{argmax}_a X_a}$ is $\rho\, \mathbb{E}[\max_a X_a]$.
   4. Check parts (a) and (c) by modifying the simulation of Section 15.4.3.1: use two actions, and draw the second estimate as `rho * est + np.sqrt(1 - rho**2) * noise` with fresh standard normal `noise`, for $\rho \in \{0, 0.5, 0.9\}$.

5. code **Replay reuse and buffer capacity.** ⊠ Once the buffer holds `warmup` transitions, `train_dqn` takes one gradient step on `batch_size` $= 128$ uniformly sampled transitions every `train_freq` $= 2$ environment steps. With the default capacity, no transition is evicted.

   1. Show that a transition stored at step $t_0 \geq 1000$ is sampled about $64\ln(50{,}000/t_0)$ times in expectation over the run, and evaluate this for $t_0 \in \{1000, 25{,}000, 49{,}000\}$. Data from which phase of training dominate the updates?

2. With a capacity of 500 transitions, how often is a transition sampled in expectation while it stays in the full buffer? Compare this with part (a).
3. Set `buffer_size` to 500 and to 5000, and lower `warmup` to 500 for the smaller capacity, because `train_dqn` waits until the buffer holds `warmup` transitions. Train one seed for each capacity and compare the return curve and the start-state value trace with the default run. At which stage of training does the damage appear?
4. A small buffer both correlates the transitions of a minibatch and discards old states. To separate the two effects, keep the default capacity and change `ReplayBuffer.sample` to return `batch_size` consecutive transitions from a uniformly drawn start index. Compare this run with the run at capacity 500. Which of the two effects accounts for the damage of part (c)?

6. code **The sync period.** ▣ `train_dqn` copies the online weights into the target network every `sync_every` $= 250$ environment steps, which is every 125 gradient steps.
   1. Section 15.4.2.4 bounds every value after n synchronizations by $(1 - \gamma^n)/(1 - \gamma)$. Evaluate this bound at the end of the budget for `sync_every` $\in \{10, 50, 250, 1000, 5000\}$, counting the synchronizations that follow the first gradient step.
   2. Rerun the DQN arm with these five values, two seeds each, and record the greedy evaluation after the budget and the start-state value trace. Which end of the sweep behaves like the run without a target network? At which end does the bound of part (a) limit what the run can reach?
   3. The bound assumes that every value starts at zero and that the online network fits its targets exactly between synchronizations. With `sync_every` $= 250$, record $\max_a Q_w(s_0, a)$ just before each of the first 40 synchronizations of one seed and plot it against the bound. Where does the run first cross the bound, and which of the two assumptions does the crossing violate?

7. **n-step targets from replay.** ▣ Several successors of DQN replace the one-step target Equation 15.36 by an n-step target, the n-step return of Section 15.1 with a maximum at its end,

$$y_t^{(n)} = \sum_{i=0}^{n-1} \gamma^i r_{t+i} + \gamma^n \max_{a'} Q_{w^-}(s_{t+n}, a'), \tag{15.38}$$

   written here for an episode that continues past step $t + n$.
   1. `ReplayBuffer.sample` returns isolated transitions. What must the buffer store, and how must `sample` choose its indices, so that $y_t^{(n)}$ can be formed?
   2. Write $y_t^{(n)}$ for an episode that terminates at step $t + j$ with $j < n$. How does it change when the episode is truncated at step $t + j$ instead?
   3. The actions $a_{t+1}, \dots, a_{t+n-1}$ were chosen by the ϵ-greedy behavior policy of an older network. For $n = 2$ and $Q_{w^-} = Q^*$, write $\mathbb{E}[y_t^{(2)} \mid s_t, a_t]$ and show that it equals the one-step target of Q^* only if a_{t+1} is greedy for Q^* with probability one. Which policy's value does the n-step target estimate?
   4. The n-step returns of Section 15.1 did not raise the question of part (c). Why not?

8. **Prioritized sampling.** ▣ Prioritized replay (Schaul et al. 2016) samples stored transition i with probability $P(i)$ instead of $1/N$ and multiplies its loss by a weight u_i.

Let $\mathbf{g}_i$ be the gradient of the loss of transition i, so that uniform replay estimates $\bar{\mathbf{g}} = \frac{1}{N}\sum_i \mathbf{g}_i$.

1. Show that a single sampled transition with weight $u_i = 1/(NP(i))$ gives an unbiased estimate of $\bar{\mathbf{g}}$.
2. Find the distribution P that minimizes the expected squared error $\mathbb{E}\|u_i\mathbf{g}_i - \bar{\mathbf{g}}\|^2$ of this estimate. For the squared loss, $\mathbf{g}_i = e_i \nabla_w Q_w(s_i, a_i)$ with residual e_i. Under which condition is sampling in proportion to $|e_i|$ optimal?
3. Schaul et al. (2016) sample with $P(i) \propto (|e_i|+c)^p$ for a small constant $c > 0$, and use the weights $u_i = \left(NP(i)\right)^{-q}$ divided by $\max_j u_j$, with q annealed toward 1 over training; the exponents p and q are the paper's α and β. Which of the two changes to the weights of part (a) affects only the step size, and which changes the point to which the updates converge?

9. code **Baird with a target network.** ⊠⊠ DQN computes its bootstrap from a frozen copy of the weights. In the computation of Baird's counterexample in Section 15.4.1.3, compute `delta` from a copy `w_target` as `gamma * (Phi @ w_target)[6] - Phi @ w`, and set `w_target = w.copy()` every k sweeps.

   1. Run 100,000 sweeps for $k \in \{1, 10, 30, 100, 300, 1000\}$ and plot both the sup norm of $\mathbf{w}$ and the largest value error $\max_i |(\Phi\mathbf{w})_i|$. For which k does the iteration converge, and how does its speed depend on k? Why does the sup norm of $\mathbf{w}$ not approach zero where the values do?
   2. Show that the seven rows of `Phi` are linearly independent, so that every vector of target values is represented exactly by some $\mathbf{w}$. Deduce that if the regression between two synchronizations were solved exactly, each synchronization would map the values $\mathbf{v}$ to $\gamma P\mathbf{v}$ under every positive state weighting. Use this to explain both ends of part (a).
   3. In the counterexample of Tsitsiklis and Van Roy, two nonterminal states have values w and $2w$ for one weight w, and every reward is zero. The first state moves to the second, and the second stays where it is with probability $1-\xi$ and terminates otherwise. Weight both states equally. Show that exact least-squares regression to the targets of a frozen weight w_k gives $w_{k+1} = \gamma(6 - 4\xi)\, w_k/5$, and that the semi-gradient iteration without a target network diverges under the same condition on γ and ξ.
   4. Both examples contain all three ingredients of the triad. Which property of the function class separates the example that a target network repairs from the one that it cannot repair?

   Adapted from Richard S. Sutton and Andrew G. Barto, Reinforcement Learning: An Introduction, Example 11.1.

10. [extended] **LayerNorm instead of a target network.** ⊠⊠ Gallici et al. (2025) prove that a sufficiently wide Q-network with LayerNorm, trained with an ℓ_2 penalty on its final-layer weights, satisfies a stability condition for temporal-difference learning without a target network, even on off-policy data. In their experiments, LayerNorm alone sufficed in every environment except Baird's counterexample.

    1. Insert a LayerNorm after each hidden linear layer of `make_qnet`, before the ReLU, and rerun the arm without a target network on three seeds. Compare the start-state value traces and the greedy evaluations with both arms of Section 15.4.2.4.
    2. Add the same layers to the DQN arm, which keeps its target network. What does

LayerNorm change there?

3. If a seed of part (a) still diverges, add an ℓ_2 penalty on the weights of the last linear layer and find the smallest coefficient that prevents divergence. What does a large coefficient do to the learned values?
4. The theorem concerns sufficiently wide networks and a regularized update. Which of its assumptions does your experiment satisfy? Which outcomes of parts (a) to (c) would contradict the paper's claim, and which would only fall outside its scope?

15.5 Soft Actor-Critic

At the end of Section 14.8 a policy network learned to swing a pendulum up and balance it, but only after a few hundred thousand environment steps. Each batch of 1,600 steps drove one update and was then thrown away. A simulator makes such steps cheap. A robot arm does not: every step takes real time, wears the hardware and risks breaking something. This section asks how an agent can learn the same controller from a few thousand steps.

Two ideas we have already met carry most of the gain. Replay, the buffer of past transitions that Section 15.4 trained from, lets every transition drive many updates instead of one, provided a critic can learn from actions that an older policy chose. The pathwise gradient of Section 14.8 then moves the actor by differentiating that critic through the action itself. DDPG (Lillicrap et al. 2016) combined the two with a deterministic actor and proved brittle, because an actor that climbs its critic also climbs the critic's errors. TD3 (Fujimoto et al. 2018) traced much of this to the maximization bias of Section 15.4.3.1 and trained two critics, trusting the lower one. Soft Actor-Critic (SAC) (Haarnoja, Zhou, Abbeel, et al. 2018; Haarnoja, Zhou, Hartikainen, et al. 2018) keeps both repairs and adds a third idea, the entropy reward of Section 15.3. The policy is paid for staying random, so an actor trained against its own critic keeps exploring instead of collapsing onto whichever action the critic currently prefers.

In one sentence, SAC adds the policy's entropy to the reward, learns the resulting value off-policy from replayed transitions, and moves the actor by differentiating the critic through its own sampled action. Two details need care. A torque confined to $[-2, 2]$ needs a bounded policy with a correct density, because the gradient now flows through the action. And the weight on entropy must suit the scale of the reward, so SAC learns it instead of fixing it. We derive the evaluation and improvement steps of the entropy-regularized objective, build and check the bounded policy, assemble the update, and then measure how many steps the pendulum needs and how far the critic's predictions can be trusted.

```
import copy
import gymnasium as gym
```

```
torch.set_num_threads(1)
```

The running example is Pendulum from Section 14.8. A pendulum hangs from a motorized pivot. The agent observes $(\cos\vartheta, \sin\vartheta, \dot\vartheta)$, with ϑ the angle from upright, and applies a torque $u \in [-2, 2]$. Each step costs $\vartheta^2 + 0.1\,\dot\vartheta^2 + 0.001\,u^2$, so a pendulum hanging still at the bottom, $\vartheta = \pi$, costs $\pi^2 \approx 9.9$ per step, more than in any other resting position. An episode lasts 200 steps. Uniformly random torques return about $-1{,}200$, and a controller

that swings up and balances returns about -200 or better. Every run below takes at most 20,000 environment steps, and every configuration is trained with three seeds; $c = 2$ is the action scale. Ranges of steps and statements about how often something happens across runs were checked over twenty runs per configuration.

```
num_env_steps, num_seeds, c = 20_000, 3, 2.0
```

15.5.1 The Maximum-Entropy Objective

15.5.1.1 Entropy as Reward

Section 15.3 kept a policy near a fixed reference policy π_{ref} by subtracting $\beta\, D_{\text{KL}}(\pi \| \pi_{\text{ref}})$ from the expected reward, where $\beta > 0$ weighs divergence against reward. When the reference is uniform over $|\mathcal{A}|$ actions, $D_{\text{KL}}(\pi \| \text{uniform}) = \log|\mathcal{A}| - H(\pi)$, with $H(\pi) = -\mathbb{E}_{a\sim\pi}[\log \pi(a)]$ the entropy of the policy. The penalty is then an entropy bonus plus a constant that no policy can change. The same holds for Pendulum's interval of torques: the uniform density on $[-2, 2]$ has entropy $\log 4 \approx 1.39$ nats, the largest of any density on that interval, and $\log 4$ takes the place of $\log|\mathcal{A}|$. SAC applies the bonus at every step:

$$J(\pi) = \mathbb{E}_{\pi}\Big[\sum_{t} \gamma^t \big(r_t + \beta\, H(\pi(\cdot \mid s_t))\big)\Big]. \tag{15.39}$$

The agent collects β units of reward for every nat of entropy at every state it visits, so the *temperature* β has units of reward per nat. We keep the letter β of Section 15.3. The SAC papers write α, which in this book is the step size of temporal-difference learning.

Unlike the optional entropy bonus in the actor loss of Section 15.2, this term is part of the objective. It changes the optimum, and the best policy stays random wherever several actions are nearly as good. For continuous actions H is a differential entropy, the integral of $-\pi \log \pi$ over a density, and it is negative for a density concentrated on an interval shorter than one. Maximum-entropy reinforcement learning is older than deep learning (Ziebart et al. 2008); SAC is its off-policy actor-critic.

Like the other methods of this chapter, SAC alternates two kinds of steps, as generalized policy iteration does (Figure 14.8): evaluation moves the value estimates toward those of the current policy, and improvement moves the policy toward the actions those estimates favor. The entropy changes both steps. A state with two actions shows how before any network appears.

15.5.1.2 Two Actions by Hand

Take one state with two actions whose values are $Q = (1, 0)$, and let $\beta = 0.5$. Section 15.3 showed that the KL-regularized optimum multiplies the reference by $e^{Q/\beta}$ and renormalizes, and that its value is $\beta \log Z$. With a uniform reference,

$$\pi^\star(a) = \frac{e^{Q(a)/\beta}}{Z}, \qquad Z = \sum_a e^{Q(a)/\beta}, \qquad \max_{\pi} \left(\mathbb{E}_{\pi}[Q] + \beta H(\pi)\right) = \beta \log Z.$$

Here $\pi^\star = (e^2, 1)/(e^2 + 1) = (0.88, 0.12)$. Its entropy is 0.37 nats and $\mathbb{E}_{\pi^\star}[Q] = 0.88$, so the soft value is $0.88 + 0.5 \cdot 0.37 = 1.06$, which equals $0.5 \log(e^2 + 1)$. The soft value exceeds

the hard maximum 1: the tilted policy gives up 0.12 of expected value and gains 0.18 in entropy bonus. A second fact matters for what follows. The bracket $Q(a) - \beta \log \pi^\star(a)$ takes the same value for both actions, $1 + 0.5 \cdot 0.13$ and $0 + 0.5 \cdot 2.13$, both 1.06. At the optimum, one sampled action gives the soft value without error. The following cell compares $\pi^\star$ with the uniform policy and with a nearly greedy one:

```
Q, beta = np.array([1.0, 0.0]), 0.5
pi_star = np.exp(Q / beta) / np.exp(Q / beta).sum()   # the tilted optimum
for name, pi in (('tilted', pi_star), ('uniform', np.array([0.5, 0.5])),
                 ('nearly greedy', np.array([0.99, 0.01]))):
    bracket = Q - beta * np.log(pi)        # one sampled action's soft value
    print(f'{name:>13}: pi = {pi.round(2)}, E[Q] + beta H = '
          f'{pi @ bracket:.3f}, bracket = {bracket.round(2)}')
print(f'beta log Z = {beta * np.log(np.exp(Q / beta).sum()):.3f}')
```

```
      tilted: pi = [0.88 0.12], E[Q] + beta H = 1.063, bracket = [1.06 1.06]
     uniform: pi = [0.5 0.5], E[Q] + beta H = 0.847, bracket = [1.35 0.35]
nearly greedy: pi = [0.99 0.01], E[Q] + beta H = 1.018, bracket = [1.01 2.3 ]
beta log Z = 1.063
```

The tilted policy attains $\beta \log Z = 1.063$, and both other policies fall short of it, the nearly greedy one at 1.018 and the uniform one at 0.847. Under them the bracket differs between the two actions, by one unit for the uniform policy, so a single sampled action gives a noisy estimate of their soft value. Only at the optimum is the estimate exact.

15.5.1.3 Soft Policy Evaluation

The soft value of a state counts the entropy collected there along with the future reward, $V^\pi(s) = \mathbb{E}_{a \sim \pi}\big[Q^\pi(s,a) - \beta \log \pi(a \mid s)\big]$, where $Q^\pi(s,a)$ includes the entropy bonuses of all later states but not that of s itself. The critic of Section 15.1 regressed onto the one-step target $r + \gamma V(s')$, cut to r at a terminal state. SAC's critic learns Q, so it needs the value of the next state in terms of Q, and it gets it from the identity above with one sampled action. Let Q_{w^-} be a slowly changing copy of the critic Q_w, the target network of Section 15.4, which keeps the regression target from moving with every update. The target is

$$y = r + \gamma \left(1 - \mathbf{1}(s' \text{ terminal})\right)\Big(Q_{w^-}(s', \tilde{a}') - \beta \log \pi_\theta(\tilde{a}' \mid s')\Big), \qquad \tilde{a}' \sim \pi_\theta(\cdot \mid s'). \tag{15.40}$$

The one new term, $-\beta \log \pi_\theta(\tilde{a}' \mid s')$, is the entropy bonus at s', estimated through the action just drawn. The transition (s, a, r, s') comes from the replay buffer, but $\tilde{a}'$ is drawn fresh from the current policy, because V^π is an expectation under the policy being evaluated. The buffer supplies what happened; the policy supplies what it would do next.

A single draw estimates the soft value of the current policy, so this step is soft policy evaluation. It is not the log-sum-exp backup of Section 15.3, which computes the value of the tilted optimum. The two agree only when the current policy is that optimum, where the bracket is the same for every action, as in the two-action case. Soft evaluation is ordinary policy evaluation for a reward augmented by the entropy bonus, so the argument of Section 14.2 applies: each application of the evaluation operator shrinks the largest error by the factor γ, and repeated application converges to Q^π. The argument needs bounded augmented

rewards, so the entropies must be bounded as well as the rewards. That holds for finitely many actions, and for continuous policies whose density stays bounded, which we assume.

15.5.1.4 Soft Policy Improvement

The improvement step also comes from Section 15.3, which showed that the regularized objective of any policy falls short of the optimum's value by a KL divergence:

$$\mathbb{E}_{\pi}[r] - \beta\, D_{\mathrm{KL}}(\pi \| \pi_{\mathrm{ref}}) = \beta \log Z - \beta\, D_{\mathrm{KL}}(\pi \| \pi^{\star}), \tag{15.41}$$

where $\pi^{\star} \propto \pi_{\mathrm{ref}}\, e^{r/\beta}$ is the tilted optimum. With r replaced by $Q(s,\cdot)$ and a uniform reference, maximizing $\mathbb{E}_{\pi}[Q] + \beta H(\pi)$ at a state is the same as minimizing $D_{\mathrm{KL}}(\pi \| e^{Q/\beta}/Z)$, since the two differ by $\beta \log Z$, which does not depend on π. A network policy usually cannot represent the tilted distribution, so SAC moves it toward the member of its family closest in this divergence, the *reverse-KL projection*. In this direction the divergence penalizes any mass the policy puts where $e^{Q/\beta}$ is small, so when the tilted distribution has two separated peaks a Gaussian settles on one of them instead of spreading over both (Section 15.3.3.5). Writing the action as a differentiable function of noise gives the actor loss

$$L_{\pi}(\theta) = \mathbb{E}_{s\sim\mathcal{D},\, z}\Big[\beta \log \pi_{\theta}(\tilde{a}_{\theta}(s,z) \mid s) - Q_{w}(s, \tilde{a}_{\theta}(s,z))\Big], \qquad \tilde{a}_{\theta}(s,z) = c\tanh\left(\mu_{\theta}(s) + \sigma_{\theta}(s)\, z\right), \tag{15.42}$$

with $z \sim \mathcal{N}(0, I)$ and states drawn from the replay buffer $\mathcal{D}$. The loss is the negative soft value of a fresh action, so lowering it raises the critic's value and the entropy together; the squash $c\tanh$, which keeps the action in range, is the subject of the next part. Its gradient is the pathwise gradient of Section 14.8. For a Gaussian action $a = \mu + \sigma z$, that gradient is $\nabla_{\mu}\mathbb{E}[Q(a)] = \mathbb{E}[Q'(\mu + \sigma z)]$: it differentiates through the action instead of weighting a score, and on that section's example its single-sample variance was 1.0, against 21.75 for the score function. It needs a critic that is differentiable in the action. With discrete actions one sums the expectation over the actions instead.

If each state's maximization were exact, the improvement step would raise the soft value everywhere.

Proposition. Let rewards and entropies be bounded, and let the family Π contain π_{old} and allow the policy at each state to be chosen independently, as a table does. If $\pi_{\mathrm{new}}(\cdot \mid s)$ maximizes $\mathbb{E}_{a\sim\pi}\big[Q^{\pi_{\mathrm{old}}}(s,a)\big] + \beta H(\pi(\cdot \mid s))$ over Π at every s, then $V^{\pi_{\mathrm{new}}} \geq V^{\pi_{\mathrm{old}}}$ everywhere.

Proof. Write H_t for the entropy of π_{new} at s_t. Because $\pi_{\mathrm{old}} \in \Pi$, optimality at $s_0 = s$, followed by $Q^{\pi_{\mathrm{old}}}(s,a) = r(s,a) + \gamma\, \mathbb{E}[V^{\pi_{\mathrm{old}}}(s_1)]$, gives

$$V^{\pi_{\mathrm{old}}}(s) \leq \mathbb{E}_{a\sim\pi_{\mathrm{new}}}\big[Q^{\pi_{\mathrm{old}}}(s,a) - \beta \log \pi_{\mathrm{new}}(a \mid s)\big] = \mathbb{E}_{\pi_{\mathrm{new}}}\big[r_0 + \beta H_0 + \gamma\, V^{\pi_{\mathrm{old}}}(s_1)\big].$$

Applying the same inequality to $V^{\pi_{\mathrm{old}}}(s_1)$, then to $V^{\pi_{\mathrm{old}}}(s_2)$, and so on n times, gives $V^{\pi_{\mathrm{old}}}(s) \leq \mathbb{E}_{\pi_{\mathrm{new}}}\big[\sum_{t<n} \gamma^t (r_t + \beta H_t)\big] + \gamma^n\, \mathbb{E}\big[V^{\pi_{\mathrm{old}}}(s_n)\big]$. Values are bounded, so the last term vanishes as $n \to \infty$, and the sum tends to $V^{\pi_{\mathrm{new}}}(s)$. ■

SAC takes one gradient step on each loss while the critic is still learning, so the proposition describes the exact algorithm, not the loop we run; Haarnoja, Zhou, Abbeel, et al. (2018) give the full argument in their Appendix B.

15.5.2 The Squashed Gaussian Policy

15.5.2.1 Clipping and the Pathwise Gradient

The Gaussian policy of Section 14.8 drew an unbounded torque and let the environment clip it to $[-2, 2]$. That did no harm to the score-function estimator, which never differentiates through the action. SAC's pathwise estimator does, and clipping causes two problems. Outside the interval the clip's derivative is zero, so a sample beyond a boundary gives the actor no gradient, even at states where the boundary torque is the best action. And clipping moves all the probability beyond each boundary onto the boundary itself, so the clipped distribution has no density there and $\log \pi$ in Equation 15.40 and Equation 15.42 is undefined. A bounded distribution with a smooth, invertible map from a Gaussian removes both problems.

15.5.2.2 The Tanh Change of Variables

Draw $u \sim \mathcal{N}(\mu_\theta(s), \sigma_\theta(s)^2)$ and squash it, $a = c \tanh u$. The map is smooth and increasing, so a short interval around u and its image around a have the same probability, and the image is shorter by the factor $da/du = c\,(1 - \tanh^2 u)$. The density of a is therefore the Gaussian density divided by this factor, coordinate by coordinate:

$$\log \pi(a \mid s) = \sum_i \Big[\log \mathcal{N}(u_i;\, \mu_i, \sigma_i) - \log\left(1 - \tanh^2 u_i\right) - \log c \Big]. \tag{15.43}$$

Each term of the sum corrects the Gaussian log-density for the compression of one coordinate. Figure 15.11 shows the effect for one mean and spread. Near $a = 2$ the squash packs the whole upper tail of u into a short interval of actions, so the density rises there. Without the correction the expression misses this compression and integrates to 1.13 instead of 1.

The $\log c$ term does not change the actor's gradient, but it shifts every log-density, and with it the reported entropy and the term $\beta \log \pi$ of the target, by $\log 2 \approx 0.69$ nats. We keep it, so that the reported entropies are those of the actual torque distribution.

The policy network is the `GaussianHead` of Section 14.8. A *trunk* computes features, a linear layer maps them to the mean $\mu_\theta(s)$, and with `state_std=True` a second linear layer gives $\log \sigma_\theta(s)$, clamped to $[-5, 2]$. SAC needs a spread that depends on the state, because its entropy bonus is paid at every state: a pendulum balancing at the top needs a narrow policy, while one being swung up can use a wide one, and a single shared σ cannot provide both. For the trunk we use two ReLU layers of 64 units, the shape of the critics below, in place of the single tanh layer of Section 14.8; published SAC implementations use the same shape with 256 units. The clamp never binds in our runs, where $\log \sigma$ stays above about -3.5, and removing it does not change their outcome. We keep it as a guard: it keeps σ away from zero, so that $\log \pi$ in the target cannot blow up through a collapsing spread. It does not bound the entropy of the squashed action, which a mean far past the squash still drives down. Common implementations clamp to $[-20, 2]$.

`SquashedGaussianPolicy` wraps the head. `sample` draws $u = \mu + \sigma z$ and returns the squashed action with its log-probability, both differentiable, for the losses. `log_prob_u` evaluates Equa-

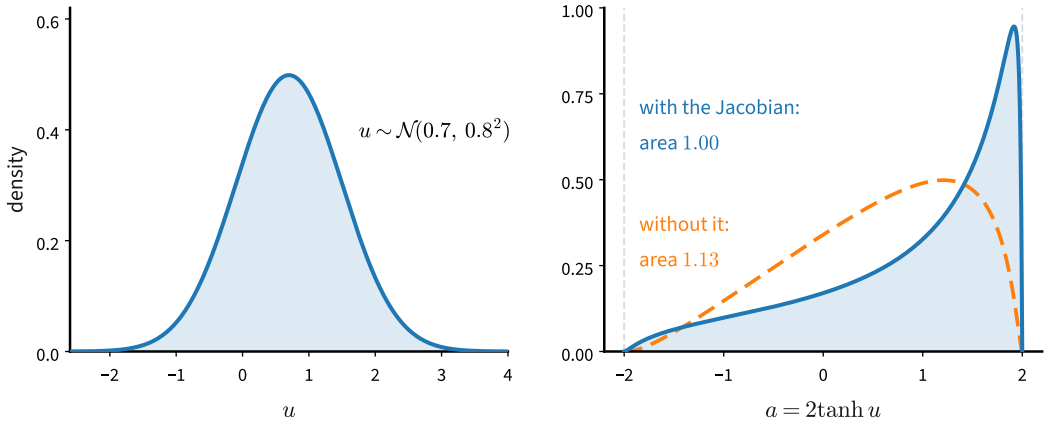


Figure 15.11: The squashed Gaussian. (a) The density of $u \sim \mathcal{N}(0.7, 0.8^2)$. (b) The density of the action $a = 2\tanh u$. With the Jacobian term of Equation 15.43 the density integrates to 1 and rises near $a = 2$, where the squash compresses the upper tail of u into a short interval. Without it, the expression is the Gaussian density read at $\operatorname{arctanh}(a/2)$, which integrates to 1.13 and misses that pile-up.

tion 15.43 from the pre-squash u that the sampler keeps, so no inverse hyperbolic tangent appears on the differentiable path. `act` draws a NumPy action for the environment, `act_logp` returns such an action with its log-probability, for the measurements at the end of this section, and both take their sample u from `_draw`; `act_greedy` returns the squashed mean $c\tanh\mu_\theta(s)$. The class is saved to the library because the next section reuses it.

```
class SquashedGaussianPolicy(nn.Module):
    """d2l.GaussianHead with SAC's network (two ReLU layers and a
    state-dependent standard deviation), squashed through
    a = scale * tanh(u)."""
    def __init__(self, obs_dim, act_dim, hidden=64, scale=2.0, seed=None):
        super().__init__()
        with d2l.seeded(seed):
            trunk = nn.Sequential(nn.Linear(obs_dim, hidden), nn.ReLU(),
                                  nn.Linear(hidden, hidden), nn.ReLU())
            self.policy = d2l.GaussianHead(obs_dim, act_dim, hidden,
                                           state_std=True, trunk=trunk)
        self.scale = scale

    def log_prob_u(self, u, mean, std):
        """log pi at a = scale tanh(u), from the pre-squash u the sampler
        keeps."""
        logdet = 2 * (np.log(2) - u - nn.functional.softplus(-2 * u))
        return (torch.distributions.Normal(mean, std).log_prob(u)
                - logdet - np.log(self.scale)).sum(-1)

    def sample(self, obs):
        """A reparameterized action and its log-probability, differentiable."""
        mean, std = self.policy(obs)
```

```
        u = mean + std * torch.randn_like(std)
        return self.scale * torch.tanh(u), self.log_prob_u(u, mean, std)

    @torch.inference_mode()
    def _draw(self, obs, rng):
        """A pre-squash sample u with the head's mean and std, from numpy
        states and the noise of rng."""
        mean, std = self.policy(torch.as_tensor(obs))
        noise = rng.standard_normal(mean.shape, dtype=np.float32)
        return mean + std * torch.as_tensor(noise), mean, std

    def act(self, obs, rng):
        """A sampled action, numpy in and out."""
        return self.scale * np.tanh(self._draw(obs, rng)[0].numpy())

    @torch.inference_mode()
    def act_logp(self, obs, rng):
        """A sampled action and its log-probability, numpy in and out."""
        u, mean, std = self._draw(obs, rng)
        return (self.scale * np.tanh(u.numpy()),
                self.log_prob_u(u, mean, std).numpy())

    @torch.inference_mode()
    def act_greedy(self, obs, rng=None):
        """The squashed mean scale * tanh(mean), numpy in and out."""
        mean = self.policy(torch.as_tensor(obs))[0].numpy()
        return self.scale * np.tanh(mean)
```

Inside `log_prob_u` the Gaussian and Jacobian terms are summed over the action coordinates before any average over the batch. Summing one and averaging the other gives the right answer for Pendulum's single coordinate and a wrong one for every larger action space.

15.5.2.3 A Numerically Stable Log-Determinant

The direct form `log(1 - tanh(u)**2)` subtracts two nearly equal numbers once $\tanh u$ is close to 1, and it fails when float32 rounds $\tanh u$ to exactly 1. Since $1 - \tanh^2 u = 4e^{-2u}/(1 + e^{-2u})^2$, taking logarithms gives

$$\log\left(1 - \tanh^2 u\right) = 2\left(\log 2 - u - \operatorname{softplus}(-2u)\right), \tag{15.44}$$

an identity that holds for every u and involves no subtraction of nearly equal numbers; `log_prob_u` uses it. The following float32 calculation compares the direct form, a common guarded version that adds 10^{-6}, and the stable form:

```
u = np.float32([0, 3, 8, 10, 20])
t = np.tanh(u)
with np.errstate(divide='ignore'):
    naive = np.log(1 - t ** 2)
guarded = np.log(1 - t ** 2 + np.float32(1e-6))
stable = 2 * (np.log(np.float32(2)) - u - np.logaddexp(np.float32(0), -2 * u))
print(f'{"u":>4} {"naive":>10} {"guarded":>10} {"stable":>10}')
```

```
for row in zip(u, naive, guarded, stable):
    print(f'{row[0]:4.0f} {row[1]:10.4f} {row[2]:10.4f} {row[3]:10.4f}')
```

```
   u      naive    guarded     stable
   0     0.0000     0.0000     0.0000
   3    -4.6187    -4.6186    -4.6187
   8   -14.5561   -13.4256   -14.6137
  10       -inf   -13.8155   -18.6137
  20       -inf   -13.8155   -38.6137
```

The direct form is already slightly off at $u = 8$, where float32 has few digits left for $1 - \tanh^2 u$, and it is $-\infty$ from $u = 10$ on. The guard avoids the infinity, but it is more than a nat too high at $u = 8$ and stays at $\log 10^{-6} = -13.8155$ from $u = 10$ on, so it no longer registers further saturation. The reported entropy and the entropy term of Equation 15.42 then go wrong while the return curve looks normal.

Normalization gives a second, independent check of the change of variables. We evaluate the density on a fine grid of actions, invert the squash on that grid, and integrate Equation 15.43 by the trapezoid rule, with and without the Jacobian term:

```
def squashed_logp(a, mu, sigma, logdet=True):
    up = np.arctanh(a / c)          # only here: the training path keeps u
    lp = (-0.5 * ((up - mu) / sigma) ** 2 - np.log(sigma)
          - 0.5 * np.log(2 * np.pi))
    if logdet:
        lp -= 2 * (np.log(2) - up - np.logaddexp(0, -2 * up)) + np.log(c)
    return lp

a_grid = np.linspace(-c + 1e-6, c - 1e-6, 200_001)
for m, s in ((0.0, 0.5), (0.7, 0.8)):
    w = np.trapezoid(np.exp(squashed_logp(a_grid, m, s)), a_grid)
    wo = np.trapezoid(np.exp(squashed_logp(a_grid, m, s, False)), a_grid)
    print(f'mu = {m}, sigma = {s}: integrates to {w:.6f} with the '
          f'log-det, {wo:.3f} without')
```

```
mu = 0.0, sigma = 0.5: integrates to 1.000000 with the log-det, 1.653 without
mu = 0.7, sigma = 0.8: integrates to 1.000000 with the log-det, 1.132 without
```

With the correction the density integrates to 1.000000 at both settings; without it the integrals are 1.65 and 1.13. This error would barely change the return on Pendulum while making every reported entropy wrong, and the check finds it without any training. The grid stops 10^{-6} short of $\pm c$, where arctanh is infinite, which is why the check uses moderate σ.

15.5.3 The Algorithm

15.5.3.1 Twin Critics and Polyak Targets

The maximum of noisy estimates is biased upward (Section 15.4.3.1): two independent estimates of the same value, each off by zero-mean noise of standard deviation σ, have a maximum that exceeds the value by about 0.56σ on average. An actor that climbs its critic's gradient does the same thing continuously. It moves toward actions whose value the critic happens to overestimate, and the bootstrap carries the error into the next target. Follow-

ing TD3, SAC trains two critics Q_{w_1} and Q_{w_2} from different initializations and uses their pointwise minimum in the target,

$$y = r + \gamma\,(1 - \mathbf{1}(s' \text{ terminal}))\Big(\min_{j=1,2} Q_{w_j^-}(s', \tilde{a}') - \beta \log \pi_\theta(\tilde{a}' \mid s')\Big), \tag{15.45}$$

and in the actor loss Equation 15.42, where $\min_j Q_{w_j}$ replaces Q_w. Both critics regress onto the same target y. Where they disagree, the minimum believes the lower one, so it is a deliberately pessimistic estimate. At the end of this section we measure what this pessimism costs and what it prevents.

Each critic has a target copy w_j^-, as in DQN, but the copy is not overwritten every 250 environment steps as in Section 15.4. After every update it moves a small fraction ν of the way toward the online weights:

$$w_j^- \leftarrow \nu\, w_j + (1 - \nu)\, w_j^-, \qquad \nu = 0.005.$$

This *Polyak averaging* is an exponential moving average with a half-life of $\ln 2/\nu \approx 139$ updates, so the regression target drifts slowly instead of standing still and then jumping. Only the critics have copies. The action $\tilde{a}'$ in the target comes from the current policy, and the target needs no extra smoothing of that action, because a stochastic policy averages its own targets over its spread. Figure 15.12 shows how the pieces of one update connect.

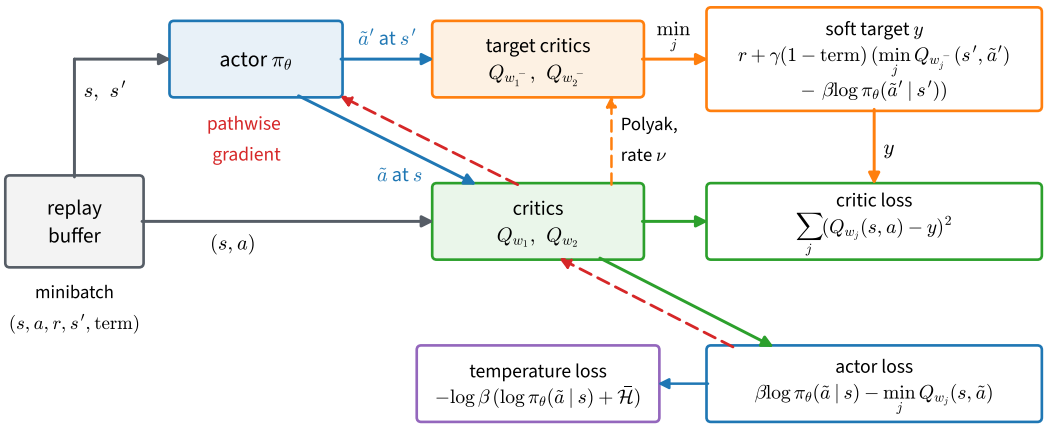


Figure 15.12: One SAC update. Top row: the soft target y, computed by the target critics at a fresh action $\tilde{a}'$ drawn from the current actor at s' (the minimum of the two, less $\beta \log \pi_\theta(\tilde{a}' \mid s')$). Middle row: the critics regress $Q_{w_j}(s, a)$ onto y. Bottom row: the actor loss reaches the actor through the critics and the fresh action $\tilde{a}$ (red, the pathwise gradient), and the temperature has a loss of its own on the same log-probability; the dashed orange arrow is the Polyak update, the only edge that writes into the target critics.

15.5.3.2 Automatic Temperature

A fixed β fixes an exchange rate between reward and entropy, and the right rate depends on the units of the reward. Multiplying every reward by 10 has the same effect on the optimal

policy as dividing β by 10 (exercise 3), so a temperature tuned for one task is wrong for the same task measured in other units. The second SAC paper (Haarnoja, Zhou, Hartikainen, et al. 2018) fixes the entropy instead. It asks for the best policy whose entropy at the visited states is at least a target $\bar{H}$,

$$\max_{\pi} \ \mathbb{E}_{\pi}\Big[\sum_{t} \gamma^{t} r_{t}\Big] \quad \text{subject to} \quad \mathbb{E}_{s}\big[H(\pi(\cdot \mid s))\big] \geq \bar{H}, \tag{15.46}$$

and treats β as the Lagrange multiplier of the constraint. The common choice is $\bar{H} = -\dim \mathcal{A}$, here -1 nat. The multiplier is learned by gradient descent on

$$J(\beta) = -\log \beta \ \mathbb{E}_{s\sim\mathcal{D},\, a\sim\pi_{\theta}}\big[\log \pi_{\theta}(a \mid s) + \bar{H}\big], \tag{15.47}$$

whose derivative with respect to $\log \beta$ is $\hat{H} - \bar{H}$, the excess of the batch's entropy estimate $\hat{H} = -\mathbb{E}[\log \pi_{\theta}]$ over the target. A step therefore lowers β while the policy is more random than the target, which makes randomness cheaper to give up, and raises β while it is less random. We learn $\log \beta$ so that β stays positive, start from $\beta = 1$, the default of common implementations, and use Adam at the learning rate of the networks, 3×10^{-4}. Adam moves a parameter by about its learning rate per step while the gradient keeps its sign, so as long as the entropy stays above the target, $\log \beta$ falls by about 3×10^{-4} per update. After 6,000 updates that predicts $\beta \approx e^{-1.8} \approx 0.17$, which we will compare with the measured trace.

The first SAC paper already trained two critics, together with a separate network for the state value. The algorithm built here has no value network and learns its temperature, as in the second paper; it is the version common implementations use.

15.5.3.3 The Update Step and the Loop

The replay buffer of Section 15.4, `d2l.ReplayBuffer`, keeps transitions $(s, a, r, s', \text{term})$ in preallocated arrays. `add` stores one transition and overwrites the oldest once the buffer is full, and `sample` draws a batch uniformly and returns it as a `d2l.Batch`. Its action column holds integers. `ReplayBufferC` widens that column to float vectors and changes nothing else. Like the policy, it is saved because the next section reuses it.

```
class ReplayBufferC(d2l.ReplayBuffer):
    """d2l.ReplayBuffer with the action column widened to float vectors."""
    def __init__(self, capacity, obs_dim, act_dim):
        super().__init__(capacity, obs_dim)
        self.act = np.zeros((capacity, act_dim), np.float32)
```

SAC learns off-policy without importance ratios. The critic target Equation 15.45 does not depend on the policy that collected the transition, because the next action is drawn fresh, and the actor loss draws its own actions too. Replay still changes the distribution of states over which the losses are averaged, since buffer states were visited by older policies, and no ratio of action probabilities corrects for that.

The `ActorCritic` container of Section 14.8, a policy with one state-value network and its optimizer, does not fit SAC, which has action-value critics, their target copies and a temperature. `SoftActorCritic` holds the actor, a list of critics and a list of target copies, $\log \beta$, and one Adam optimizer each for the actor, the critics and $\log \beta$. With `beta=None` the temperature is

learned, starting from $\beta = 1$; a number fixes it. `beta` returns the temperature as a scalar array. `min_q` evaluates the minimum over a list of critics, the online ones by default, and `q_np` does the same on NumPy inputs for the measurements at the end.

```
class SoftActorCritic:
    """One actor, a list of action-value critics, their frozen copies, and
    the temperature beta = exp(log_beta), learned if beta is None."""
    def __init__(self, actor, qs, targets, lr, beta=None):
        self.actor, self.qs, self.targets = actor, qs, targets
        self.opt_pi = torch.optim.Adam(actor.parameters(), lr=lr)
        self.opt_q = torch.optim.Adam(
            [p for q in qs for p in q.parameters()], lr=lr)
        start = 1.0 if beta is None else beta   # a learned beta starts at 1
        self.log_beta = torch.tensor(float(np.log(start)),
                                     requires_grad=beta is None)
        self.opt_beta = (torch.optim.Adam([self.log_beta], lr=lr)
                         if beta is None else None)

    @property
    def beta(self):
        """The temperature exp(log_beta), a scalar tensor without gradient."""
        return self.log_beta.detach().exp()

    def min_q(self, obs, act, nets=None):
        """min_j Q_j(s, a) over the online critics, or over `nets`."""
        x = torch.cat([obs, act], -1)
        return torch.min(torch.stack(
            [q(x).squeeze(-1) for q in (self.qs if nets is None else nets)]),
            0).values

    def q_np(self, obs, act):
        """min_j Q_j(s, a), numpy in and out."""
        with torch.inference_mode():
            return self.min_q(torch.as_tensor(obs),
                              torch.as_tensor(act)).numpy()
```

`make_critic` builds one critic, a network from the concatenated state and action to one number, with two ReLU layers of 64 units. `make_sac` builds an agent whose target copies start equal to their critics.

```
def make_critic(in_dim):
    """Q(s, a) from the concatenated state and action."""
    return nn.Sequential(nn.Linear(in_dim, 64), nn.ReLU(),
                         nn.Linear(64, 64), nn.ReLU(), nn.Linear(64, 1))

def make_sac(seed, obs_dim, act_dim, num_critics=2, lr=3e-4, beta=None):
    """A seeded agent whose target critics start as copies of the critics."""
    torch.manual_seed(seed)
    actor = SquashedGaussianPolicy(obs_dim, act_dim)
    qs = [make_critic(obs_dim + act_dim) for _ in range(num_critics)]
    return SoftActorCritic(actor, qs, [copy.deepcopy(q) for q in qs], lr,
                           beta)
```

One update follows Figure 15.12, and three functions hold its losses. `soft_target` computes the target Equation 15.45 at a fresh action from the live policy; it is data for the regression, so no gradient flows into it. `critic_loss` regresses every critic onto that one target, and `actor_loss` is Equation 15.42 with the minimum of the critics, at a fresh reparameterized action; it also returns the log-probabilities, which the temperature's loss reuses.

```
def soft_target(agent, rew, next_obs, term, beta, gamma):
    """eq_sac_target_twin at a fresh action from the live policy."""
    with torch.no_grad():                    # the critic target is data
        a2, logp2 = agent.actor.sample(next_obs)
        return rew + gamma * (1 - term) * (
            agent.min_q(next_obs, a2, agent.targets) - beta * logp2)

def critic_loss(qs, obs, act, y):
    """Squared error of every critic against the one shared target y."""
    x = torch.cat([obs, act], -1)
    return sum(((q(x).squeeze(-1) - y) ** 2).mean() for q in qs)

def actor_loss(actor, min_q, obs, beta):
    """beta log pi - min_j Q_j at a fresh reparameterized action; also
    returns log pi."""
    a, logp = actor.sample(obs)
    return (beta * logp - min_q(obs, a)).mean(), logp
```

`sac_step` performs the update. The critics take one step on `critic_loss`, then the actor takes one step on `actor_loss`. If the agent learns β, the temperature takes one step on $J(\beta)$ with the actor's log-probabilities. Finally the target critics move by ν, called `polyak` in the code. The step returns the batch mean of $\log \pi_\theta$, whose negative estimates the policy's entropy, and leaves it on the device, so that the training loop does not wait for every update.

```
def descend(opt, loss):
    """One optimizer step on a loss."""
    opt.zero_grad()
    loss.backward()
    opt.step()

def sac_step(agent, obs, act, rew, next_obs, term, gamma, polyak,
             target_entropy):
    """One SAC update: soft critic regression, pathwise actor step, Polyak
    averaging, and, if the agent learns beta, one temperature step toward
    the target entropy. Returns the batch mean of log pi."""
    beta = agent.beta
    y = soft_target(agent, rew, next_obs, term, beta, gamma)
    descend(agent.opt_q, critic_loss(agent.qs, obs, act, y))
    loss_pi, logp = actor_loss(agent.actor, agent.min_q, obs, beta)
    descend(agent.opt_pi, loss_pi)
    if agent.opt_beta is not None:   # entropy above target: beta falls
        descend(agent.opt_beta, -agent.log_beta * (logp.detach().mean()
                                                   + target_entropy))
    with torch.no_grad():                    # Polyak: the drifting copy
        for q, tnet in zip(agent.qs, agent.targets):
            for pt, pq in zip(tnet.parameters(), q.parameters()):
```

```
                pt.lerp_(pq, polyak)
    return logp.detach().mean()
```

`descend` takes one optimizer step on a loss; `lerp_` moves each target weight a fraction `polyak` of the way toward its critic's.

`train_sac` runs one agent on Pendulum. The first `warmup` steps take uniformly random torques; after that the actor acts, and every environment step is followed by `utd` updates. This *update-to-data ratio* is the number of gradient steps taken per environment step. The loop stores `terminated`, not `truncated`. Pendulum never terminates and its episodes end only at the 200-step time limit, so the stored flag is always zero and the critic bootstraps across the cut, as a time limit requires (Section 14.8). At the end of each episode the loop yields the step count, the episode's return, the mean of $-\log \pi_\theta$ over the episode's updates as an estimate of the entropy, and the current β. The update ratio and the buffer are arguments so that every experiment below uses this one loop, which is the off-policy loop of `train_dqn` in Section 15.4 with continuous actions. `make_update` binds `sac_step` to the agent and turns a sampled `d2l.Batch` into arrays. It also seeds the run's sampling noise, so a run gives the same result whether it runs alone or after others.

```
def make_update(agent, seed, **hp):
    """sac_step bound to the agent, on a d2l.Batch; seeds this run's
    sampling noise."""
    torch.manual_seed(seed)
    def update(b):
        return sac_step(agent, *map(torch.as_tensor, (
            b.obs, b.act, b.rew, b.next_obs, b.term)), **hp)
    return update
```

```
def train_sac(seed, agent, buffer, num_env_steps, utd=1, warmup=1_000,
              batch_size=256, gamma=0.99, polyak=0.005, target_entropy=-1.0):
    """SAC on Pendulum: `warmup` uniformly random steps, then `utd` updates
    per environment step. Yields (env step, episode return, policy
    entropy, beta) at the end of every episode."""
    update = make_update(agent, seed, gamma=gamma, polyak=polyak,
                         target_entropy=target_entropy)
    rng, env = np.random.default_rng(seed), gym.make('Pendulum-v1')
    scale, act_dim = agent.actor.scale, buffer.act.shape[1]
    obs, ep_return = env.reset(seed=seed)[0], 0.0
    logp_sum, n_upd = 0.0, 0
    for t in range(1, num_env_steps + 1):
        a = (rng.uniform(-scale, scale, act_dim).astype(np.float32)
             if t <= warmup else agent.actor.act(obs, rng))
        next_obs, rew, terminated, truncated, _ = env.step(a)
        buffer.add(obs, a, rew, next_obs, float(terminated))
        obs, ep_return = next_obs, ep_return + rew
        for _ in range(utd if t >= warmup else 0):
            logp_sum = logp_sum + update(buffer.sample(batch_size, rng))
            n_upd += 1
        if terminated or truncated:   # read the device sum once per episode
            yield (t, ep_return, float(-logp_sum / n_upd) if n_upd else np.nan,
```

```
                  float(agent.beta))
            obs, ep_return = env.reset()[0], 0.0
            logp_sum, n_upd = 0.0, 0
```

`run_arm` trains one configuration on several seeds and keeps the agents for later measurements. Its buffer holds every transition of a run. We start with two configurations of twin critics and a learned temperature: one update per environment step for 20,000 steps, and ten updates per step for 4,000 steps.

```
def run_arm(seeds, num_env_steps, utd=1, num_critics=2, beta=None):
    """Train one SAC configuration per seed. Returns the agents and, per
    seed, an array of per-episode rows (step, return, entropy, beta)."""
    agents = [make_sac(seed, 3, 1, num_critics, beta=beta) for seed in seeds]
    runs = [np.array(list(train_sac(seed, agent,
                                    ReplayBufferC(num_env_steps, 3, 1),
                                    num_env_steps, utd=utd)))
            for seed, agent in zip(seeds, agents)]
    return agents, runs

seeds, agents, runs = range(num_seeds), {}, {}
agents['UTD 1'], runs['UTD 1'] = run_arm(seeds, num_env_steps)
agents['UTD 10'], runs['UTD 10'] = run_arm(seeds, 4_000, utd=10)
```

15.5.4 Sample Efficiency and Calibration

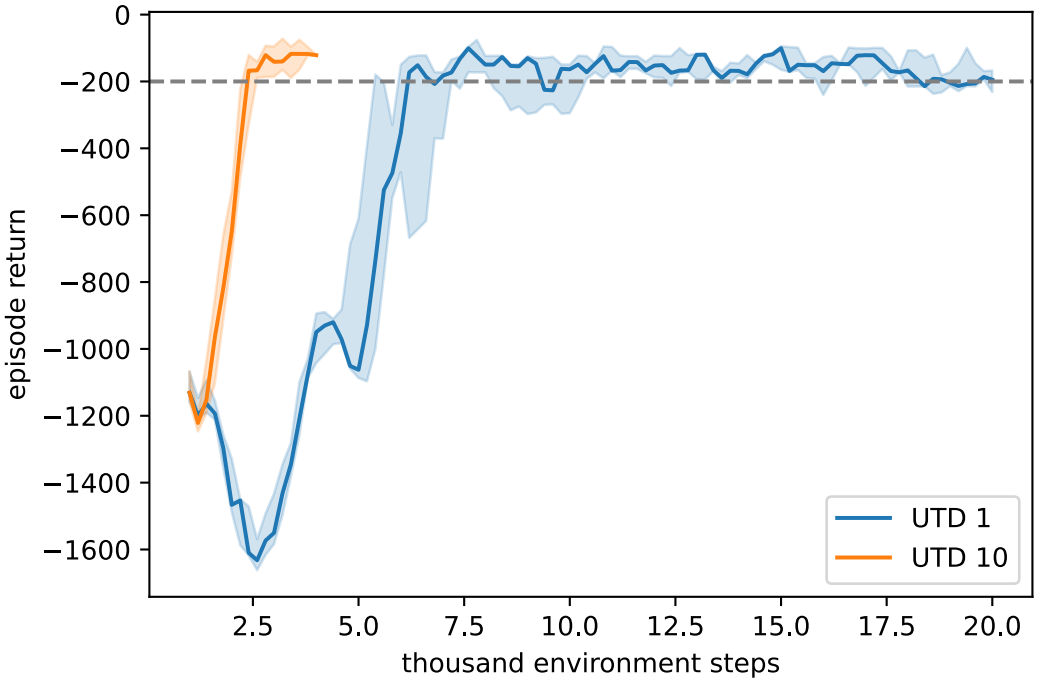
15.5.4.1 Return against Environment Steps

The plot shows each episode's return, averaged over the last five episodes, against environment steps, with -200 marked as the level of a controller that swings up and balances. `crossings` lists, per seed of an arm, the first step at which a run's five-episode average reaches that level, found by `d2l.steps_to` of Section 15.1; `inf` would mark a run that never does.

```
def crossings(arm):
    """Per seed of an arm, the env step at which the five-episode average
    return first reaches -200."""
    return ', '.join(f'{d2l.steps_to(r, -200, 5):.0f}' for r in runs[arm])

grid = np.arange(1_000, num_env_steps + 1, 200)
curves = {arm: d2l.on_grid(runs[arm], grid, 1, smooth=5) for arm in runs}
curves['UTD 10'][:, grid > 4_000] = np.nan      # those runs end at 4,000
d2l.plot_curves(curves, x=grid / 1000, xlabel='thousand environment steps',
                ylabel='episode return', reference=-200)
for arm in runs:
    print(f'{arm}: steps to a five-episode average of -200, per seed '
          f'{crossings(arm)}')
```

```
UTD 1: steps to a five-episode average of -200, per seed 6200, 5400, 6000
UTD 10: steps to a five-episode average of -200, per seed 2400, 2400, 2600
```



At one update per environment step every seed crosses -200 after about five to eleven thousand environment steps. REINFORCE with a learned baseline crossed the same level on the same task only after a few hundred thousand steps (Section 14.8), one to two orders of magnitude more. Three changes account for the difference. Replay lets every environment step drive a gradient step, where REINFORCE spent 1,600 steps on one update. The bootstrapped target lets the critic learn from single transitions instead of whole returns. And the pathwise gradient carries the critic's knowledge of how the value changes with the torque straight into the actor.

Before the curves rise, they fall. Episodes under the random warm-up torques return about $-1{,}200$; once the actor takes over, the returns drop to about $-1{,}600$ between 1,500 and 3,000 steps, worse than random, before they climb. Two behaviors could cost that much. An actor could spin the pendulum and incur the speed cost, or it could hold the pendulum at the bottom and incur the largest angle cost. To tell them apart we measure what a policy does to the pendulum: `behavior` runs ten episodes and reports the mean return, the mean of $\cos\vartheta$, which is -1 at the bottom and 1 at the top, and the mean speed $|\dot{\vartheta}|$. Three fixed controllers serve as references: zero torque, a spinning controller that always pushes along the motion with torque $2\,\mathrm{sign}(\dot{\vartheta})$, and a damping controller $-2\tanh(2\dot{\vartheta})$ that opposes every motion.

```
def behavior(policy, num_episodes=10, seed=0):
    """Mean return, cos(theta) and |theta dot| over a policy's episodes."""
    env, rng = gym.make('Pendulum-v1'), np.random.default_rng(seed)
    env.reset(seed=seed)
    ret, states = 0.0, []
    for _ in range(num_episodes):
        obs, done = env.reset()[0], False
```

```
            while not done:
                obs, rew, terminated, truncated, _ = env.step(policy(obs, rng))
                done, ret = terminated or truncated, ret + rew
                states.append(obs)
        s = np.array(states)
        return ret / num_episodes, s[:, 0].mean(), np.abs(s[:, 2]).mean()

    policies = {'zero torque': lambda obs, rng: np.zeros(1, np.float32),
                'spin': lambda obs, rng: 2 * np.sign(obs[2:]),
                'damp': lambda obs, rng: -2 * np.tanh(2 * obs[2:])}
```

We compare them with each seed's actor after 2,000 steps, in the middle of the dip, and with a trained agent. Because every run sets its own seed, training a seed again for 2,000 steps retraces the start of its run above. We measure the actor's mean action $c \tanh \mu_\theta(s)$:

```
    for seed in seeds:
        agent = make_sac(seed, 3, 1)
        for _ in train_sac(seed, agent, ReplayBufferC(2_000, 3, 1), 2_000):
            pass
        policies[f'SAC at 2k, seed {seed}'] = agent.actor.act_greedy
    policies['SAC at 20k, seed 0'] = agents['UTD 1'][0].actor.act_greedy
    print(f'{"policy":>18} {"return":>8} {"cos theta":>10} {"|theta dot|":>12}')
    for name, policy in policies.items():
        ret, cos, speed = behavior(policy)
        print(f'{name:>18} {ret:8.0f} {cos:10.2f} {speed:12.2f}')
```

```
            policy   return  cos theta  |theta dot|
       zero torque    -1290      -0.49         2.50
              spin    -1610       0.04         6.83
              damp    -1886      -0.96         0.22
 SAC at 2k, seed 0    -1789      -0.93         0.55
 SAC at 2k, seed 1    -1769      -0.93         0.69
 SAC at 2k, seed 2    -1784      -0.93         0.63
SAC at 20k, seed 0     -167       0.83         0.58
```

After 2,000 steps every actor keeps the pendulum near the bottom, with $\cos\vartheta$ about -0.9, and nearly still, with a speed of about one radian per second or less. It behaves like the damping controller, which scores the worst of the three references, while the trained agent keeps the pendulum near the top. The spinning controller earns a similar return with a very different motion, fast and with $\cos\vartheta$ near zero, which none of the actors shows. So the actor first learns to damp the swing, which leaves the pendulum hanging in its costliest position, and only later learns to pump energy into it and swing it up. On all twenty runs the actor's mean action after 2,000 steps held the pendulum at the bottom in the same way. A likely reason, which we have not tested separately: a young critic has propagated the costs only a few steps ahead, and over a few steps the speed and torque costs are what an action changes, so damping looks best.

The second curve takes ten updates per environment step. It crosses -200 after 2,400 to 4,000 environment steps, against 5,400 to 11,200 at one update per step, about 2.5 times fewer. Its dip comes earlier, near 1,200 steps, and is gone by 1,600. The saved environment steps cost computation: up to the crossing the higher ratio takes about three times as many gradient steps, and the whole 4,000-step run takes 30,000 of them, about 1.6 times the 19,000

of the 20,000-step run. A higher update ratio exchanges computation for interaction, the exchange a robot arm wants. Below 2,400 steps there is little left to gain this way, since the first 1,000 steps are random and a five-episode average cannot reach -200 before step 2,000.

15.5.4.2 Entropy and Temperature

To see what the learned temperature does, we compare it with a fixed $\beta = 0.2$, a value many implementations use, in three more runs at one update per environment step:

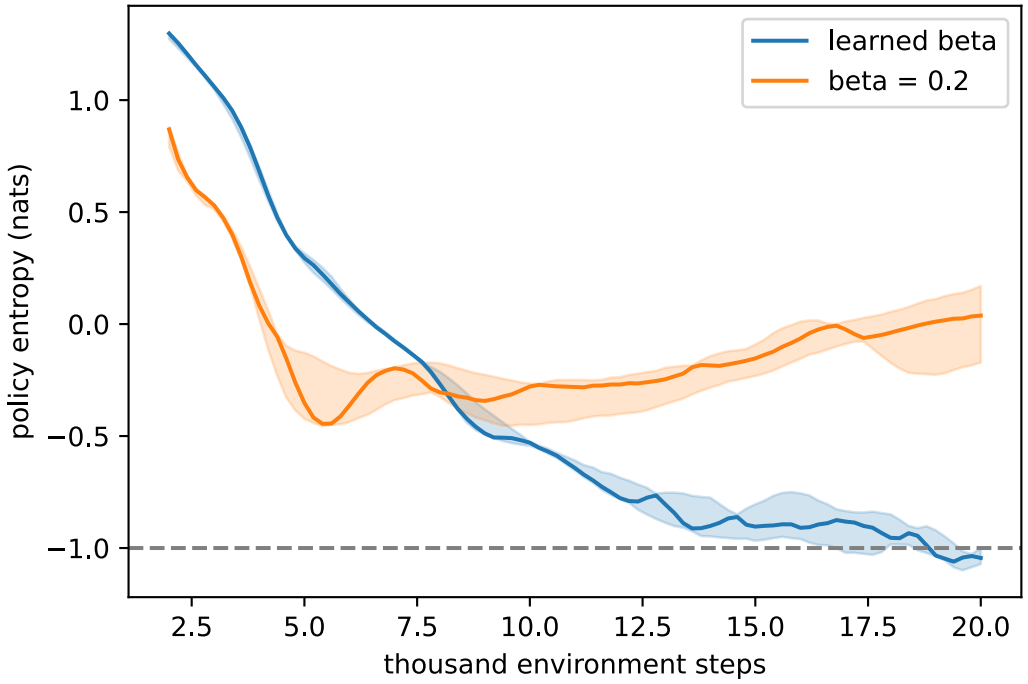
```
agents['beta = 0.2'], runs['beta = 0.2'] = run_arm(seeds, num_env_steps,
                                                   beta=0.2)
print(f'beta = 0.2: steps to -200, per seed {crossings("beta = 0.2")}')
```

```
beta = 0.2: steps to -200, per seed 6200, 6800, 6400
```

The fixed temperature crosses -200 in the same range of steps as the learned one, and neither crossed consistently earlier. The entropy estimates of both, against the target of -1 nat:

```
pair = {'learned beta': runs['UTD 1'], 'beta = 0.2': runs['beta = 0.2']}
grid = np.arange(2_000, num_env_steps + 1, 200)
d2l.plot_curves({arm: d2l.on_grid(r, grid, 2, smooth=5)
                 for arm, r in pair.items()},
                x=grid / 1000, xlabel='thousand environment steps',
                ylabel='policy entropy (nats)', reference=-1)
for arm, r in pair.items():
    print(f'{arm}: entropy over the last 20 episodes, per seed '
          f'{np.round([np.nanmean(x[-20:, 2]) for x in r], 2)}')
```

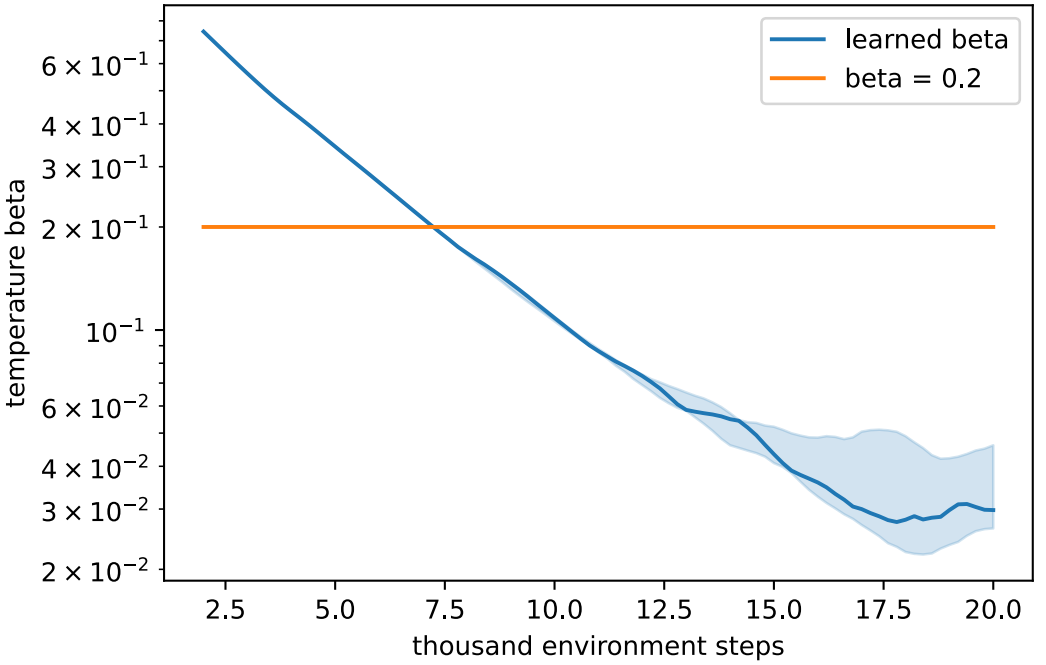
```
learned beta: entropy over the last 20 episodes, per seed [-0.97 -0.99 -0.95]
beta = 0.2: entropy over the last 20 episodes, per seed [-0.15 -0.02  0.09]
```



With the learned temperature the entropy falls steadily from above one nat onto the target and stays near it from about 15,000 steps on. At the fixed $\beta = 0.2$ it dips to a few tenths of a nat below zero while the swing-up is being learned and then rises again. Once the pendulum balances, the critic is nearly flat around the best torque, so spread costs little value: for a critic quadratic in the action with curvature h, the best Gaussian has variance β/h (exercise 3), and a flatter critic allows a wider policy. At 20,000 steps the rise is still under way, so the final entropy at a fixed temperature depends on when the run stops. The temperature of both configurations, on a logarithmic scale:

```
d2l.plot_curves({arm: d2l.on_grid(r, grid, 3) for arm, r in pair.items()},
                x=grid / 1000, xlabel='thousand environment steps',
                ylabel='temperature beta', yscale='log')
print('learned beta, median over seeds at 2k, 7k and 20k steps:',
      np.median(d2l.on_grid(pair['learned beta'], [2_000, 7_000, 20_000], 3),
                axis=0).round(3))
```

```
learned beta, median over seeds at 2k, 7k and 20k steps: [0.744 0.212 0.03 ]
```



Until about 10,000 steps the learned β falls from 1 along a straight line on the logarithmic scale: Adam's step of about the learning rate per update, taken while the entropy exceeds its target. It is near 0.2 when the runs cross -200, close to the prediction of 0.17 after 6,000 updates. Once the entropy reaches the target the fall slows, and β ends at a few hundredths. By lowering β as the critic flattens, the learned temperature removes the dip and the rebound of the entropy; it does not change how fast the pendulum is learned.

Training returns come from the stochastic policy, because entropy is part of what that policy maximizes. We also evaluate the mean action $c \tanh \mu_\theta(s)$ of every trained agent, on twenty episodes each:

```
env = gym.make('Pendulum-v1')
for arm in runs:
    greedy_return, cost = [], []
    for seed, agent in zip(seeds, agents[arm]):
        env.reset(seed=100 + seed)
        g = d2l.evaluate(env, agent.actor.act_greedy, num_episodes=20)
        env.reset(seed=100 + seed)
        s = d2l.evaluate(env, agent.actor.act, num_episodes=20,
                         rng=np.random.default_rng(1000 + seed))
        greedy_return.append(g)
        cost.append(g - s)
    print(f'{arm:>10}: mean action {np.round(greedy_return)}, '
          f'cost of sampling {np.round(cost, 1)}')
```

```
     UTD 1: mean action [-188. -147. -184.], cost of sampling [-5.9  0.2  0.1]
    UTD 10: mean action [-186. -139. -185.], cost of sampling [0.3 0.1 0. ]
beta = 0.2: mean action [-179. -142. -184.], cost of sampling [ 7.2  6.6 -0.8]
```

Every agent's mean action returns between about -120 and -190, and sampling from the policy instead changes the return by at most about ten points. The learned, state-dependent spread keeps the policy random at little cost in return.

15.5.4.3 Critic Calibration

The actor climbs its critic, so the critic's errors become the actor's. To see whether the minimum of two critics does what it was built for, we compare the prediction at the first state of an episode with what the policy then collects. For each agent we run twenty stochastic episodes and record $\min_j Q_{w_j}(s_0, a_0)$ beside the realized discounted soft return, $r_0 + \sum_{t \geq 1} \gamma^t (r_t - \beta \log \pi_\theta(a_t \mid s_t))$, which is what a soft critic predicts. The realized return also counts $-\beta \log \pi_\theta(a_0 \mid s_0)$, whose mean is β times the entropy at s_0, below 0.1 in size for these agents. It stops at step 200, while the critic bootstraps across the time limit and predicts a return without end; the missing tail is $\gamma^{200} \approx 0.13$ times the value at step 200, which is near zero for a balanced pendulum.

For a critic without the minimum we train single-critic agents at ten updates per environment step, the setting in which the twin critics above learned fastest:

```
agents['single'], runs['single'] = run_arm(seeds, 4_000, utd=10,
                                           num_critics=1)
print(f'single critic, UTD 10: steps to -200, per seed '
      f'{crossings("single")}')
```

```
single critic, UTD 10: steps to -200, per seed 2800, 2600, 2600
```

The single critic reaches -200 in the same range of steps as the twin critics at this ratio, so the return does not tell the two apart. The calibration does:

```
def calibration(agent, seed, gamma=0.99, num_episodes=20):
    """Mean predicted min_j Q(s0, a0) and mean realized discounted soft
    return, at the agent's temperature beta."""
    rng, env = np.random.default_rng(seed), gym.make('Pendulum-v1')
    env.reset(seed=seed)
    beta, preds, softs = float(agent.beta), [], []
    for _ in range(num_episodes):
        obs, done, disc, soft = env.reset()[0], False, 1.0, 0.0
        a, logp = agent.actor.act_logp(obs, rng)
        preds.append(float(agent.q_np(obs, a)))
        while not done:
            obs, rew, terminated, truncated, _ = env.step(a)
            done = terminated or truncated
            soft += disc * (rew - beta * logp)
            disc *= gamma
            if not done:
                a, logp = agent.actor.act_logp(obs, rng)
        softs.append(soft)
    return np.mean(preds), np.mean(softs)

for name, arm in (('twin, UTD 1', 'UTD 1'), ('twin, UTD 10', 'UTD 10'),
                  ('single, UTD 10', 'single')):
```

```
    for seed, agent in zip(seeds, agents[arm]):
        pred, soft = calibration(agent, 200 + seed)
        print(f'{name:>14}, seed {seed}: predicted {pred:7.1f}, '
              f'realized {soft:7.1f}, gap {pred - soft:+6.1f}')
```

```
   twin, UTD 1, seed 0: predicted  -154.6, realized  -114.3, gap  -40.3
   twin, UTD 1, seed 1: predicted  -113.2, realized   -94.8, gap  -18.4
   twin, UTD 1, seed 2: predicted  -172.0, realized  -138.7, gap  -33.3
  twin, UTD 10, seed 0: predicted  -136.7, realized  -116.4, gap  -20.3
  twin, UTD 10, seed 1: predicted   -88.3, realized   -93.1, gap   +4.8
  twin, UTD 10, seed 2: predicted  -158.9, realized  -140.4, gap  -18.5
single, UTD 10, seed 0: predicted   -68.1, realized  -119.7, gap  +51.6
single, UTD 10, seed 1: predicted   -57.5, realized   -94.9, gap  +37.4
single, UTD 10, seed 2: predicted  -105.9, realized  -143.4, gap  +37.5
```

The twin minimum sits below the realized return on almost every run, and at most a few points above it on the others. The single critic predicts more than the twin minimum on every run and more than the realized return on most. Over the twenty runs per configuration these signs held as stated, while the sizes of the gaps varied from run to run by tens of points. The minimum therefore prevents the overestimation that a single critic shows here, while the minimum itself predicts too little; on Pendulum neither changes the return. Whether a higher update ratio makes a single critic overestimate more is the subject of exercise 9.

The pendulum is now learned from a few thousand environment steps, and a higher update ratio exchanges computation for further environment steps. Each transition still teaches the critic one number, the target of one regression, and even at ten updates per step the swing-up took a few thousand transitions. A person who has pushed a swing a few times can predict where it will go next. Section 15.6 asks whether an agent that learns the pendulum's physics from its transitions can plan its way up from a few hundred.

15.5.5 Summary

SAC maximizes reward plus entropy with an off-policy actor-critic. Soft policy evaluation adds $-\beta \log \pi$ to the critic's bootstrap target, and soft policy improvement moves the actor toward the reverse-KL projection of $e^{Q/\beta}$ by a pathwise gradient through the critic. A tanh squash keeps actions bounded, and its log-density needs the change-of-variables term, computed in a stable form. Twin critics with Polyak-averaged copies keep the critic from overestimating, and a learned temperature holds the entropy at a target instead of fixing an exchange rate between reward and entropy. On Pendulum, SAC swings up within about five to eleven thousand environment steps at one update per step and within about three thousand at ten, one to two orders of magnitude fewer than REINFORCE.

15.5.6 Exercises

1. **Entropy in the target.** The term $-\beta \log \pi_\theta$ in Equation 15.40 makes the critic value entropy collected at later states, whereas the entropy bonus of Section 15.2 acts only in the actor loss. Consider a start state s_0 with two actions, L and R, both with reward 0. Action L leads to a state s_L with a single available action, and action R leads to a state s_R with n available actions. Every action at s_L and at s_R has reward 0 and ends the episode.

1. Under Equation 15.39, compute the optimal soft values of s_L and s_R and the optimal probability of R at s_0. Evaluate the probability for $n = 10$ and $\gamma = 0.99$. How does it depend on β?
2. Suppose that the critic target omits $-\beta \log \pi_\theta$ while the actor loss keeps $\beta \log \pi_\theta$. Which probability of R does the actor converge to, and which objective does this variant optimize?
3. The KL penalty of Section 15.3 with a reference that is uniform over the actions available at each state equals the entropy bonus up to a constant. Repeat part a with this penalty. Why does the constant change the optimal policy here, and why does it not change the optimal policy on Pendulum?

2. code **Variance of the sampled bracket.** The target Equation 15.40 estimates the soft value $\mathbb{E}_{a\sim\pi}[Q(s',a) - \beta \log \pi(a \mid s')]$ from a single action $\tilde{a}'$. Take the two actions with $Q = (1, 0)$ of the example by hand, $\beta = 0.5$, and the policies $\pi_T(a) \propto e^{Q(a)/T}$ for temperatures $T > 0$.
   1. Show that under π_T the bracket $B(a) = Q(a) - \beta \log \pi_T(a)$ has variance $(1 - \beta/T)^2 \operatorname{Var}_{\pi_T}[Q]$ and mean $\beta \log Z - \beta D_{\mathrm{KL}}(\pi_T \| \pi_\beta)$, where $Z = \sum_a e^{Q(a)/\beta}$.
   2. Evaluate the mean and the variance for $T \in \{0.1, 0.25, 0.5, 1, 2\}$. For which T does a single sample give the exact soft value?
   3. Repeat part b with $\beta = 0$, the target without entropy. For which policies does the single-sample target have zero variance then? What does the comparison imply for the noise in SAC's critic targets as the actor approaches $e^{Q/\beta}/Z$?

3. code **Quadratic critic.** Near its best action $a^\star$, let a critic be quadratic in a one-dimensional action, $Q(s,a) = Q(s,a^\star) - \frac{h}{2}(a - a^\star)^2$ with curvature $h > 0$, and let the policy be the unsquashed Gaussian $\mathcal{N}(\mu, \sigma^2)$.
   1. Maximize $\mathbb{E}_{a\sim\pi}[Q(s,a)] + \beta H(\pi)$ over μ and σ. Show that the maximizer is the tilted distribution $e^{Q(s,\cdot)/\beta}/Z$, so that the reverse-KL projection of Section 15.5.1.4 is exact for this family.
   2. Show that the optimal policy loses $Q(s,a^\star) - \mathbb{E}_{a\sim\pi}[Q(s,a)] = \beta/2$ for every h.
   3. Exercise 1c of Section 15.3 showed for a single decision that scaling the reward has the same effect as dividing the KL coefficient by the same factor. Show that, in any MDP, multiplying every reward by $k > 0$ leads to the same optimal policy under Equation 15.39 as dividing β by k. Why must a fixed β be retuned when the units of the reward change?
   4. Let $\beta \to 0$. What happens to σ? At $\beta = 0$ and $\sigma_\theta = 0$, the loss Equation 15.42 becomes $-Q_w(s, c \tanh \mu_\theta(s))$, the loss of the deterministic actor of DDPG (Lillicrap et al. 2016). Where must the exploration of such an agent come from? How does this limit relate to problem 2b of Section 14.8?
   5. Multiply every reward by 10 before it enters the buffer, keep the reported episode return unscaled, and train one seed with the fixed $\beta = 0.2$ and one with the learned temperature. Use part c to predict how the entropy of the first run and the final β of the second compare with the runs of Section 15.5.4.2, then check. Haarnoja, Zhou, Abbeel, et al. (2018) show the sensitivity of SAC with a fixed temperature to the reward scale in their Figure 3(b).

4. code **Entropy of the squashed policy.** Let $u \sim \mathcal{N}(\mu, \sigma^2)$ and $a = c \tanh u$ with $c = 2$, as in `SquashedGaussianPolicy`.
   1. Use Equation 15.43 to show that

$$H(\pi) = \tfrac{1}{2}\log(2\pi e\sigma^2) + \mathbb{E}_u\big[\log(1-\tanh^2 u)\big] + \log c. \tag{15.48}$$

At $\mu = 0$, which term grows with σ, and which one shrinks?

2. Evaluate H at $\mu = 0$ for $\sigma \in \{0.1, 0.3, 1, 3\}$ as an expectation over u, by Monte Carlo with `log_prob_u` or by quadrature in u. Near which σ does H peak? Compare the peak with $\log 2c$, the entropy of the uniform distribution on $[-c, c]$, which no density on this interval exceeds.

3. Construct a freshly initialized `SquashedGaussianPolicy` for Pendulum. Estimate its entropy by Monte Carlo with `sample`, averaged over $1{,}000$ states visited under uniformly random torques, as in the warm-up of `train_sac`. How close does the estimate come to the value of part b at $\mu = 0$ and $\sigma = 1$? Which outputs of the fresh network on these states explain any difference? If the actor loss kept only its term $\beta \log \pi_\theta$, which value would the entropy trace of Section 15.5.4.2 approach?

Adapted from Berkeley CS 285, Assignment 3, Section 3.3.

5. **Off-policy without ratios.** The buffer of `train_sac` keeps every transition of a run, including the $1{,}000$ uniformly random warm-up actions, whereas Equation 15.40 and Equation 15.42 draw fresh actions from the current policy.

   1. A variant of `sac_step` replaces $\tilde{a}'$ by the action that the collecting policy took next, at s'. Which policy's value does the critic then estimate, and what does the term $-\beta \log \pi_\theta(a' \mid s')$ measure at such an action? Which ratio would correct the target, and why can it not be computed from this buffer?
   2. Both losses average over states drawn from the buffer rather than from the current policy. The Proposition of Section 15.5.1.4 improves the policy at every state. Why does the weighting of the states not matter for exact per-state maximization? Why does it matter for an actor network of limited capacity?

6. **Projection direction.** ⊠ SAC moves its actor toward the reverse-KL projection of $e^{Q/\beta}/Z$ onto its policy family Π. A method that fits the policy by maximum likelihood to samples of $e^{Q/\beta}/Z$ computes the forward projection $\arg\min_{\pi\in\Pi} D_{\mathrm{KL}}(e^{Q/\beta}/Z \,\|\, \pi)$ instead.

   1. The proof of the Proposition of Section 15.5.1.4 uses a single property of π_{new} at each state. State it, and use Equation 15.41 to show that the reverse-KL projection onto any family that contains π_{old} has this property.

   2. Consider a single state with $\gamma = 0$, so that the soft value of a policy is $\mathbb{E}_\pi[Q] + \beta H(\pi)$. Let

$$Q(a) = \beta \log\Big(\tfrac{1}{2}\,\mathcal{N}(a; -m, 1) + \tfrac{1}{2}\,\mathcal{N}(a; m, 1)\Big), \tag{15.49}$$

   let Π be the one-dimensional Gaussians, and let $\pi_{\text{old}} = \mathcal{N}(m, 1)$. Show that the forward projection is $\mathcal{N}(0, 1 + m^2)$. Evaluate the soft values of π_{old} and of the forward projection numerically, in units of β, for $m \in \{1, 2, 3, 4\}$. For which m does the forward projection lower the soft value?

   3. Explain part b with the coverage behavior of the two KL directions described in Section 15.3.3.5. Which direction does the improvement guarantee require? Why does the forward projection fail only when the modes of $e^{Q/\beta}$ are well separated?

7. code **Restoring force of the Jacobian.** ▣ In the actor loss Equation 15.42, the term $\beta \log \pi_\theta$ contains $-\beta \log(1 - \tanh^2 u)$ for each action coordinate, with $u = \mu_\theta(s) + \sigma_\theta(s)z$.
   1. Use Equation 15.44 to show that this term contributes $2\beta \tanh(u)\, \partial u/\partial \theta$ to the gradient of the loss. In which direction does it move u, and how strong can this push become?
   2. With automatic differentiation in float32, evaluate the derivative with respect to u of the stable form and of the guarded expression `log(1 - tanh(u)**2 + 1e-6)` at $u \in \{3, 5, 7, 10, 20\}$, and compare both with $-2 \tanh u$. Beyond which $|u|$ does an actor trained with the guarded expression lose the push of part a?
   3. Delete `logdet` from `log_prob_u`. Show that the remaining entropy term changes the loss at the rate $-\beta/\sigma$ per unit of σ and does not depend on μ. Why can the critic term of the loss not hold σ back once u saturates? Which entropy does the trace report if σ reaches the clamp e^2, and how does that value compare with the bound of problem 4b?
   4. Retrain one seed with the deleted term. Record the return curve, the entropy trace, and the quadrature check of `squashed_logp` applied to the modified density. Which of the three readings expose the error?

8. code **Twin-critic disagreement.** ▣ The calibration of Section 15.5.4.3 found that the minimum of two critics predicts less than a single critic does.
   1. Problem 4a of Section 15.4 wrote the maximum of two numbers through their mean and their distance. Write the minimum the same way, and conclude that the target Equation 15.45 lies below the target built from the average of the two target critics, at a nonterminal s', by $\frac{\gamma}{2}\left|Q_{w_1^-}(s', \tilde{a}') - Q_{w_2^-}(s', \tilde{a}')\right|$.
   2. Suppose that every backup lowers the target by the same amount b. Show that the fixed point of soft policy evaluation is lowered by $b/(1-\gamma)$, and evaluate this factor for $\gamma = 0.99$.
   3. For each agent in `agents['UTD 10']`, run twenty stochastic episodes as `calibration` does, and record $|Q_{w_1}(s, a) - Q_{w_2}(s, a)|$ at every visited state s, with a drawn from the policy. Predict from parts a and b how far the twin critics' prediction should lie below that of a critic without the minimum, and compare the prediction with the difference between the printed gaps of the twin and the single critics at ten updates per step.
   4. Which effects does the constant offset of part b leave out?

9. code **Update ratio.** The runs of Section 15.5.4.1 took one and ten updates per environment step.
   1. Train three seeds with `utd=20` for 3,000 environment steps, with twin critics and with a single critic. Does the crossing come earlier than at ten updates per step? Repeat with `warmup=200`.
   2. Measure the agents of part a with `calibration`. Does the single critic's gap grow from ten to twenty updates per step? Does the twin minimum stay at or below the realized return?
   3. Count the gradient steps that each run takes before it crosses -200, at one, ten and twenty updates per step. What does a higher ratio exchange for environment steps, and what limits how far the exchange can go?

10. [extended] **Discrete soft actor-critic.** ▣ With a finite action set, the policy network outputs the probabilities $\pi_\theta(\cdot \mid s)$ through a softmax, and each critic outputs one

value per action, as the Q-network of Section 15.4 does (Christodoulou 2019).

1. Write the critic target and the actor loss with the expectations over $\tilde{a}'$ and $\tilde{a}$ computed exactly by summing over actions. Which parts of `SquashedGaussianPolicy` are no longer needed? Why does the actor gradient no longer require a critic that is differentiable in the action?
2. Why is the target $\bar{H} = -\dim \mathcal{A}$ of the automatic temperature in Section 15.5.3.2 meaningless for a discrete policy, and in which interval must a target for $|\mathcal{A}|$ actions lie?
3. Implement the method on CartPole with twin critics and Polyak targets, and compare its return against environment steps with `train_dqn` of Section 15.4 at the same budget and seeds.

15.6 Learned Models and Planning

Soft actor-critic swung the pendulum up after a few thousand transitions, and it used every transition in the same way: as the target of one regression for its critic, a reward plus the discounted value of the next state. A transition records more than that. It shows how the pendulum moved under a particular torque, and the physics behind that motion is the same at every visit. A child who has been pushed on a swing a few times can say where the swing will be a moment later and how hard to push to send it higher. Could an agent learn the physics of the pendulum from a few hundred steps and plan its way up, with no critic and no policy?

Section 14.3 did this on the lake. It counted the transitions it had seen into an estimated model and ran value iteration on the estimate, treating it as exact. Pendulum's states are continuous, so they cannot be counted. Regression takes the place of counting: a network learns to predict the next state and the reward from the current state and the torque. Search takes the place of value iteration: the agent tries many torque sequences inside the model and executes the first torque of the best one. Sutton's Dyna architecture combined a learned model with planning in 1990 (Sutton 1990), and neural-network models searched in this way control simulated robots from minutes of experience (Nagabandi et al. 2018).

A model lets an agent use computation in place of experience: every real step can be followed by thousands of imagined ones. A search can also start from what a policy proposes and teach its result back to the policy. AlphaZero and MuZero learned to play board games beyond human strength in this way (Silver et al. 2018; Schrittwieser et al. 2020), and search guided by a learned policy is one way language models spend extra computation at test time. In this section we fit a model of Pendulum from one random episode, measure how its errors grow along a plan, plan with it, and turn the search into a better policy. One question stays with us throughout: how far from its data can a model be trusted?

```
import gymnasium as gym
```

```
torch.set_num_threads(1)
```

15.6.1 A Dynamics Model from Data

15.6.1.1 Regression in Place of Counting

Pendulum is the task of Section 15.5. A pendulum hangs from a motorized pivot; the state is $s = (\cos\vartheta, \sin\vartheta, \dot\vartheta)$, where ϑ is the angle from upright, and the action is a torque $a \in [-2, 2]$. Each step gives the reward $-(\vartheta^2 + 0.1\,\dot\vartheta^2 + 0.001\,a^2)$ with ϑ wrapped to $[-\pi, \pi]$, and an episode lasts 200 steps. A pendulum hanging still at the bottom loses $\pi^2 \approx 9.9$ per step, about $2{,}000$ per episode; random torques lose about 900 to $1{,}700$; a controller that swings the pendulum up and balances it loses between zero and a few hundred, depending on where it starts.

A *model* of the environment predicts what the environment does. We write $\hat f_\phi$ for a network with parameters ϕ that maps a state and an action to a predicted next state and reward,

$$(\hat s', \hat r) = \hat f_\phi(s, a). \tag{15.50}$$

On the lake the model was a table of frequencies, and every entry was estimated only from the visits to its own state-action pair. A network shares its parameters across states, so a transition at one angle also teaches the model about nearby angles. We fit it by least squares on the stored transitions (s_i, a_i, r_i, s_i'),

$$\min_\phi \sum_i \left\| \hat f_\phi(s_i, a_i) - (s_i', r_i) \right\|^2. \tag{15.51}$$

The fit asks the model to reproduce the recorded next state and reward of every stored transition. With a table in place of the network, the minimizer at each pair would be the average of the outcomes observed there: the counted model of Section 14.3, reduced to its mean. Pendulum is deterministic, so the mean outcome is the outcome. A stochastic environment would need a model of the whole distribution of next states, as the counts provided on the lake.

Two choices make the regression easy. The network predicts the change of state $s' - s$ rather than s' itself, because in one step of 0.05 seconds the pendulum moves only a little, and the change is the part worth learning. And every input and target coordinate is standardized to mean zero and variance one over the stored data, because the coordinates live on different scales: the cosine and sine lie in $[-1, 1]$, the angular velocity in $[-8, 8]$, and the reward in about $[-16, 0]$. Without standardization least squares would mostly fit the reward.

The settings for the whole section come first: the torque bound $c = 2$, the depth $n = 25$ of a plan, the number of candidate plans the search tries, the number of Adam steps per fit of the model, the number of seeds, and the seed that fixes the starting states of our test.

```
c, horizon, num_cand = 2.0, 25, 500   # torque bound, depth n, candidates
fit_steps, num_seeds, test_seed = 1_000, 5, 1_000
```

Data come from whole episodes. `run_episode` acts in a *vector environment*, gymnasium's container that steps several copies of Pendulum in lockstep, so that one call can run a controller from ten starting states at once. A controller is any function from a batch of observations to a batch of actions, and `random_controller` returns one that draws uniformly random torques. `score` runs a controller for one episode from ten fixed starts, those of environment seeds

1000 to 1009, and returns the mean return; it is the test for every controller in this section. The transitions go into a `d2l.ReplayBufferC`, the replay buffer of Section 15.5 with a floating-point action column. Here it serves only as a store of arrays; we never sample batches from it.

```
def run_episode(envs, controller, buffer=None, seed=None, num_steps=200):
    """One episode in every environment of a vector env, acting by
    controller(obs) -> actions. Returns the returns; records transitions."""
    obs, total = envs.reset(seed=seed)[0], 0.0
    for _ in range(num_steps):
        act = controller(obs)
        next_obs, rew, term, _, _ = envs.step(act)
        for i in range(envs.num_envs if buffer is not None else 0):
            buffer.add(obs[i], act[i], rew[i], next_obs[i], float(term[i]))
        obs, total = next_obs, total + rew
    return total

def random_controller(rng, scale=c):
    """Uniformly random torques: the data of the first episode."""
    return lambda obs: rng.uniform(-scale, scale, (len(obs), 1)).astype(
        np.float32)

def score(controller, num_starts=10, seed=test_seed):
    """Mean return of a controller over ten fixed Pendulum starts."""
    envs = gym.make_vec('Pendulum-v1', num_envs=num_starts,
                        vectorization_mode='sync')
    return run_episode(envs, controller, seed=seed).mean()
```

`model_data` turns the stored transitions into the inputs (s, a) and targets $(s' - s, r)$ of the regression, with their means and standard deviations.

```
def model_data(buffer):
    """Inputs (s, a) and targets (s' - s, r) of the stored transitions,
    with their means and standard deviations."""
    n = buffer.size
    x = np.concatenate([buffer.obs[:n], buffer.act[:n]], -1)
    y = np.concatenate([buffer.next_obs[:n] - buffer.obs[:n],
                        buffer.rew[:n, None]], -1)
    return x, y, (x.mean(0), x.std(0) + 1e-6, y.mean(0), y.std(0) + 1e-6)
```

`DynamicsModel` is a multilayer perceptron with two hidden layers of 64 units and the smooth SiLU activation (Section 4.1). It holds the standardization statistics as part of its state, beside its parameters; its forward pass standardizes the input, undoes the standardization of the output, and adds the predicted change to the state. `fit_model` recomputes the statistics from all stored data, then takes Adam steps on minibatches of 256 transitions. Its optimizer is created once with the model, in `make_model`, so each later fit continues from the previous parameters instead of starting over.

```
class DynamicsModel(nn.Module):
    """(s', r) from (s, a): an MLP that maps standardized inputs to the
    standardized change of state and reward."""
```

```
    def __init__(self, obs_dim=3, act_dim=1, hidden=64):
        super().__init__()
        d_in, d_out = obs_dim + act_dim, obs_dim + 1
        self.net = nn.Sequential(nn.Linear(d_in, hidden), nn.SiLU(),
                                 nn.Linear(hidden, hidden), nn.SiLU(),
                                 nn.Linear(hidden, d_out))
        self.register_buffer('mu_x', torch.zeros(d_in))   # standardization
        self.register_buffer('sd_x', torch.ones(d_in))
        self.register_buffer('mu_y', torch.zeros(d_out))
        self.register_buffer('sd_y', torch.ones(d_out))

    def forward(self, obs, act):
        x = (torch.cat([obs, act], -1) - self.mu_x) / self.sd_x
        y = self.net(x) * self.sd_y + self.mu_y   # (s' - s, r) in data units
        return obs + y[..., :-1], y[..., -1]

def make_model(seed, lr=1e-3):
    torch.manual_seed(seed)
    model = DynamicsModel()
    return model, torch.optim.Adam(model.parameters(), lr=lr)

def fit_model(model, opt, buffer, rng, num_steps=fit_steps, batch_size=256):
    """Least squares on all stored transitions, in standardized units."""
    x, y, stats = model_data(buffer)
    model.mu_x, model.sd_x, model.mu_y, model.sd_y = map(torch.as_tensor,
                                                         stats)
    mu_x, sd_x, mu_y, sd_y = stats
    x = torch.as_tensor((x - mu_x) / sd_x)
    y = torch.as_tensor((y - mu_y) / sd_y)
    for _ in range(num_steps):
        i = torch.as_tensor(rng.integers(len(x), size=batch_size))
        loss = ((model.net(x[i]) - y[i]) ** 2).mean()
        opt.zero_grad()
        loss.backward()
        opt.step()
    return float(loss.detach())
```

We collect one episode of random torques from seed 0 and fit the model to its 200 transitions:

```
env = gym.make_vec('Pendulum-v1', num_envs=1, vectorization_mode='sync')
rng = np.random.default_rng(0)
buffer = d2l.ReplayBufferC(200, 3, 1)
print(f'return of the random episode: '
      f'{run_episode(env, random_controller(rng), buffer, seed=0)[0]:.0f}')
model, opt = make_model(0)
print(f'final minibatch loss (standardized units): '
      f'{fit_model(model, opt, buffer, rng):.4f}')
```

```
return of the random episode: -1072
final minibatch loss (standardized units): 0.0018
```

The random episode loses about a thousand, and the loss at the end of the fit is a small fraction of the targets' variance, which is one in standardized units.

15.6.1.2 One Step by Hand

A small training loss does not tell us what the model predicts. We look at one transition of the episode in the units of the problem. `rollout_model` feeds a state and a sequence of actions through the model, each predicted state becoming the input of the next step; with a single action it is one prediction.

```
@torch.inference_mode()
def rollout_model(model, obs, actions):
    """Open-loop rollout in the model: obs (..., 3) and actions (..., n, 1)
    give the predicted states (..., n, 3) and rewards (..., n), numpy in
    and out."""
    s, actions = torch.as_tensor(obs), torch.as_tensor(actions)
    states, rewards = [], []
    for t in range(actions.shape[-2]):
        s, r = model(s, actions[..., t, :])
        states.append(s)
        rewards.append(r)
    return torch.stack(states, -2).numpy(), torch.stack(rewards, -1).numpy()
```

```
def angle(s):
    """Angle from upright in degrees, from states (cos, sin, velocity)."""
    return np.degrees(np.arctan2(s[..., 1], s[..., 0]))

i = 100
pred, r_hat = rollout_model(model, buffer.obs[i], buffer.act[i][None])
print('s       ', buffer.obs[i].round(4), f'a = {buffer.act[i, 0]:.4f}')
print("true s' ", buffer.next_obs[i].round(4), f'r = {buffer.rew[i]:.4f}')
print("model s'", pred[0].round(4), f'r = {r_hat[0]:.4f}')
print(f'angle error {abs(angle(pred[0]) - angle(buffer.next_obs[i])):.3f} '
      f'degrees')
```

```
s        [-0.9407 -0.3394  5.6896] a = -0.0800
true s'  [-0.8154 -0.5789  5.4231] r = -11.0511
model s' [-0.8109 -0.584   5.436 ] r = -11.2768
angle error 0.386 degrees
```

At step 100 the pendulum is at $\vartheta = \operatorname{atan2}(-0.3394, -0.9407) = -2.795$ radians, about 160 degrees from upright, and it swings at 5.69 radians per second while the torque is nearly zero. The reward is $-(2.795^2 + 0.1 \cdot 5.690^2 + 0.001 \cdot 0.08^2) = -11.05$, as printed. In the 0.05 seconds of one step the angle moves to $\operatorname{atan2}(-0.5789, -0.8154) = -144.6$ degrees, about $5.6 \times 0.05 = 0.28$ radians or 16 degrees, and the model predicts this move to within a fraction of a degree. Its reward is off by less than three tenths. One step is easy: it is a small correction of a state the model has just been shown.

15.6.1.3 Compounding Error

A plan needs more than one step. To score a sequence of torques the model must predict 25 steps ahead, and from the second step on its input is its own previous prediction. The model was trained only on true states, never on its own slightly wrong ones, so an error at one step changes the input of the next: a small error in the angle changes the torque that gravity exerts, which changes the velocity, which changes the angle further. We measure

this with *open-loop* rollouts, which follow the actions recorded in an episode while the model predicts the states, and we compare the predicted angle after k steps with the recorded one.

`kstep_error` starts a rollout at every step of an episode that leaves 25 steps before the end, and reports the root-mean-square error of the angle after $k = 1, 5, 10, 25$ steps, wrapped so that $+179$ and -179 degrees are two degrees apart.

```
def kstep_error(model, buffer, first=0, ks=(1, 5, 10, 25), T=200):
    """RMS angle error (degrees) of open-loop model rollouts along the
    recorded actions, from every step of the stored T-step episodes (rows
    first onward) that leaves max(ks) steps before the episode ends."""
    n = max(ks)
    ep = lambda x: x[first:buffer.size].reshape(-1, T, *x.shape[1:])
    obs, act, nxt = ep(buffer.obs), ep(buffer.act), ep(buffer.next_obs)
    windows = lambda x: np.stack([x[:, t:t + n]
                                  for t in range(T - n + 1)], 1)
    pred = rollout_model(model, obs[:, :T - n + 1], windows(act))[0]
    d = np.radians(angle(pred) - angle(windows(nxt)))
    d = np.arctan2(np.sin(d), np.cos(d))        # wrapped to (-pi, pi]
    return [np.degrees(np.sqrt(np.mean(d[..., k - 1] ** 2))) for k in ks]
```

For each of five seeds we fit a model to one random episode and test it on a second random episode from the same seed:

```
models = []
print('seed   k=1    k=5   k=10   k=25   (degrees, held-out random episode)')
for seed in range(num_seeds):
    rng = np.random.default_rng(seed)
    train, test = d2l.ReplayBufferC(200, 3, 1), d2l.ReplayBufferC(200, 3, 1)
    run_episode(env, random_controller(rng), train, seed=seed)
    run_episode(env, random_controller(rng), test)
    m, o = make_model(seed)
    fit_model(m, o, train, rng)
    models.append((m, test))
    errors = kstep_error(m, test)
    print(f'{seed:4d} ' + ' '.join(f'{e:6.2f}' for e in errors))
```

```
seed   k=1    k=5   k=10   k=25   (degrees, held-out random episode)
   0   0.44   1.91   2.52   1.51
   1   1.83   7.08   8.67  12.29
   2   0.53   2.37   3.65   5.51
   3   0.38   1.89   4.77  11.82
   4   0.04   0.12   0.13   0.33
```

On every seed the error after ten steps is several times the error after one step, and after one step it is a fraction of a degree on most seeds. After 25 steps the error depends on the seed, but on these held-out episodes it stays below about 20 degrees: under random torques the pendulum keeps swinging near the bottom, where the training episode went too. We return to this proviso after the planner has taken the pendulum elsewhere.

The depth of a plan is a dial like the n of the n-step returns in Section 15.1. A deeper plan sees further consequences of a torque; a swing-up takes a few dozen steps, and a plan that ends before the pendulum rises cannot tell a good pumping sequence from a bad one. But each

further step adds the model's error to the prediction. We use $n = 25$ steps, 1.25 seconds, and exercise 1 moves the dial.

15.6.2 Planning in the Model

15.6.2.1 Random Shooting

Inside the model, trying a sequence of torques costs a few matrix multiplications and no real step. The simplest search uses this directly. Suppose the pendulum hangs still at the bottom. We draw five random sequences of 25 torques, roll each out in the model fitted above, and add up its predicted rewards:

```
rng = np.random.default_rng(0)
s = np.array([[-1.0, 0.0, 0.0]], np.float32)      # hanging down, at rest
A = rng.uniform(-c, c, (1, 5, horizon, 1)).astype(np.float32)
_, r_hat = rollout_model(model, np.repeat(s[:, None], 5, axis=1), A)
real = gym.make('Pendulum-v1').unwrapped          # the same sequences, for real
for a, ret in zip(A[0], r_hat[0].sum(-1)):
    real.reset()
    real.state = np.array([np.pi, 0.0])           # angle, velocity
    true = sum(real.step(a_t)[1] for a_t in a)
    print(f'first actions {a[:3, 0].round(2)}  predicted {ret:7.1f}  '
          f'true {true:7.1f}')
```

```
first actions [ 0.55 -0.92 -1.84]  predicted  -174.2  true  -237.6
first actions [-0.47  1.99  1.92]  predicted  -173.8  true  -232.7
first actions [ 1.15 -1.04  1.51]  predicted  -174.3  true  -229.4
first actions [-0.3   0.48  1.98]  predicted  -170.7  true  -223.2
first actions [-0.08 -1.07  1.21]  predicted  -174.5  true  -242.4
```

Hanging still for 25 steps would lose about $25 \times 9.9 \approx 247$, and the five predictions lie well above that and within a few points of each other. The last column replays the same five sequences in the real environment from the same state. They lose between 223 and 242: rocking the pendulum a little away from the bottom saves between 5 and 24 points against hanging still, not the 60 to 75 points the model predicts. The model is too optimistic by 40 to 70 points on every sequence. The likely cause is coverage. In the random episode the pendulum swept through the bottom fast, so the model never saw it hanging nearly still there, and its rewards for that state are too high. Even so, the sequence with the highest predicted return is also the best of the five in the real environment, and choosing among candidates needs only their ranking, not their values. Of the five we would pick that sequence and apply its first torque. How far the model can be trusted is already the question in this smallest case: here its values are wrong, and its choice is right.

That is *random shooting*: draw many candidate sequences $(a_0^{(j)}, \dots, a_{n-1}^{(j)})$, predict their returns

$$\hat{G}^{(j)} = \sum_{t=0}^{n-1} \hat{r}_t^{(j)},$$

where $\hat{r}_t^{(j)}$ is the model's reward for step t of candidate j, and keep the best. The sum is undiscounted, since over 25 steps with $\gamma = 0.99$ a discount would change the weights by

about a fifth at most. Only the first torque of the best sequence is executed. At the next step the agent observes the real state and plans again from there. This scheme, *model-predictive control* (MPC), is a long-standing method in control engineering. Replanning at every step keeps the model's errors from accumulating along the executed trajectory, because the next real observation replaces the prediction before its errors can compound; the errors over the later steps of a plan still shape which first torque wins.

`plan` carries this out for a batch of B states at once. It perturbs a given plan for each state with Gaussian noise into 500 candidates, keeps the given plan itself as one of them, clips the torques to $[-c, c]$, rolls all candidates out with `rollout_model`, and returns each state's best sequence with its predicted return. Because the given plan is one of the candidates, the returned plan never has a lower predicted return than the plan it started from. One action costs $500 \times 25 = 12{,}500$ evaluations of the model.

```
def plan(model, obs, init, rng, noise=c, num_cand=num_cand, scale=c):
    """Random shooting around a plan for each of B states: obs (B, 3), init
    (B, n, 1). Each plan is perturbed into num_cand candidates (the plan
    itself is one), all are rolled out in the model, and the candidate with
    the highest predicted return is returned with that return."""
    B = len(obs)
    A = init[:, None] + noise * rng.standard_normal(
        (B, num_cand) + init.shape[1:], dtype=np.float32)
    A[:, 0] = init
    A = np.clip(A, -scale, scale)
    s = np.repeat(obs[:, None], num_cand, axis=1)          # (B, num_cand, 3)
    ret = rollout_model(model, s, A)[1].sum(-1)            # (B, num_cand)
    best = ret.argmax(1)
    return A[np.arange(B), best], ret[np.arange(B), best]
```

15.6.2.2 Warm Starts

Which plan should the search start from? Without any knowledge it would draw fresh random sequences at every step. That throws away the plan found one step earlier, which was presumably good, and forces the search to find a swing-up among 500 random sequences again at every step. Consecutive torques then come from unrelated plans. A *warm start* reuses the previous plan: its first torque has just been executed, so the remaining 24 are shifted forward by one step, a zero torque is appended, and the search perturbs this shifted plan. A good plan is then found once and refined from step to step. Figure 15.13 shows one step of this search.

`mpc` builds such a controller from a model. The plans live in the controller's closure, one per environment, created at its first call; the perturbation has standard deviation equal to the torque bound, $c = 2$. In separate runs with five seeds, scored on another set of ten starts, we compared it with a planner that draws 500 fresh, uniformly random sequences at every step. After 600 real steps the warm-started planner scored between about -115 and -155 on every run, and the fresh planner between about -165 and -555; after 1,000 real steps the fresh planner was still 40 to 110 points worse on every seed.

```
def mpc(model, rng, horizon=horizon, noise=c):
    """Model-predictive control: at every step plan in the model, starting
```

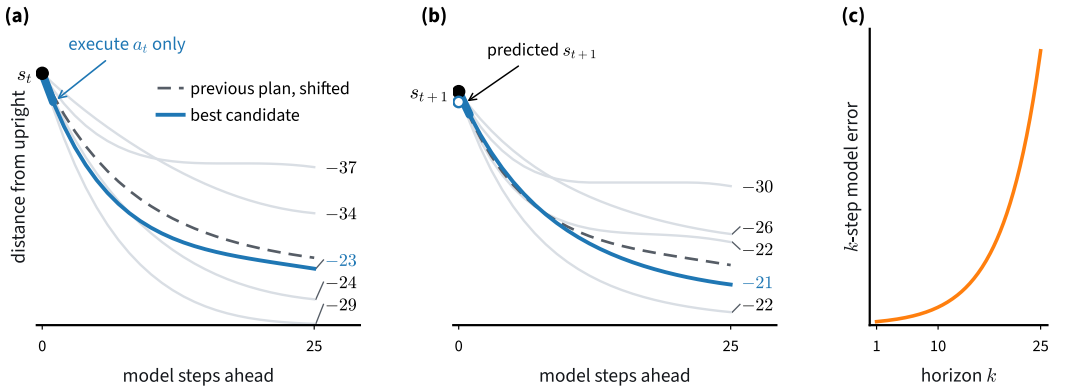


Figure 15.13: Planning in a learned model by random shooting with a warm start. (a) At state s_t, candidate action sequences drawn around the previous plan shifted by one step (dashed) are rolled out through $\hat{f}_\phi$ for 25 steps and scored by the sum of predicted rewards (numbers at the right); only the first action of the best sequence (blue) is executed. (b) The real next state differs a little from the predicted one (open circle), and the planner starts again from it, around the best plan of (a) shifted by one step. (c) The model's open-loop error grows with the horizon, one reason to replan at every step. The vertical axis and the scores come from a schematic one-dimensional model, not from Pendulum; a score adds the rewards of every step, including speed and torque costs, so the sequence that ends nearest upright need not score best.

```
    from the previous plan shifted by one step, and execute its first
    action."""
    plans = None
    def controller(obs):
        nonlocal plans
        if plans is None:
            plans = np.zeros((len(obs), horizon, 1), np.float32)
        shifted = np.concatenate([plans[:, 1:], np.zeros_like(plans[:, :1])],
                                 1)
        plans = plan(model, obs, shifted, rng, noise)[0]
        return plans[:, 0]
    return controller
```

15.6.2.3 Swing-Up from a Few Episodes

How many real steps does the planner need? `mpc_run` starts with one random episode. Then, three times, it refits the model on all data so far, scores the planner on the ten test starts, and runs one episode with the planner to collect more data; a last refit and score follow. The scores therefore measure the planner built from 200, 400, 600 and 800 real steps.

```
def mpc_run(seed, num_rounds=3):
    """One random episode, then rounds of: refit the model on all data,
    score the planner on the ten test starts, collect one MPC episode."""
```

```
    env = gym.make_vec('Pendulum-v1', num_envs=1, vectorization_mode='sync')
    rng, (model, opt) = np.random.default_rng(seed), make_model(seed)
    buffer = d2l.ReplayBufferC(200 * (num_rounds + 1), 3, 1)
    run_episode(env, random_controller(rng), buffer, seed=seed)
    scores = []
    for r in range(num_rounds + 1):
        fit_model(model, opt, buffer, rng)
        scores.append(score(mpc(model, rng)))
        if r < num_rounds:
            run_episode(env, mpc(model, rng), buffer)
    return scores, buffer
```

The printout shows the returns of the four collected episodes and the four scores, for five seeds:

```
runs = [mpc_run(seed) for seed in range(num_seeds)]
print('seed  episode returns: random, MPC x 3 |'
      '  test score at 200/400/600/800 steps')
for seed, (scores, buf) in enumerate(runs):
    rets = buf.rew[:buf.size].reshape(-1, 200).sum(1)
    print(f'{seed:4d} ' + ''.join(f'{r:7.0f}' for r in rets) + '  | '
          + ''.join(f'{s:7.0f}' for s in scores))
```

```
seed  episode returns: random, MPC x 3 |  test score at 200/400/600/800 steps
   0   -1072   -242   -118   -125  |    -181   -168   -168   -171
   1    -876   -264   -121   -123  |    -266   -201   -204   -180
   2    -966   -122   -125   -116  |    -337   -190   -168   -169
   3   -1501  -1287   -246     -2  |   -1464   -265   -191   -195
   4   -1703  -1576  -1323   -120  |   -1567  -1437   -171   -187
```

What is a good score? From the ten test starts, a controller that swings up and balances loses close to nothing from the start nearest the top and about 250 to 320 from each of the five starts near the bottom, about 170 on average. The soft actor-critic of Section 15.5, its greedy actor scored the same way after every episode, settles near -170 and first reaches -200 after about 5,000 to 7,500 steps at one update per step.

After 600 real steps, three episodes, the planner scores between about -165 and -215 on every run, close to that level, and after 800 steps between about -170 and -195. It first reaches -200 after 200 to 800 real steps, and by 600 on all runs but one. On three of the five seeds the model of the random episode alone already gives a planner that scores between about -175 and -340, far from the random policy's $-1{,}000$ and below. On the other two, seeds 3 and 4, the first planner fails, with a score near $-1{,}500$; its first one or two episodes fail as well, and after 600 real steps the same planner succeeds.

15.6.2.4 Coverage

The two slow seeds are the two whose random episode lost the most, about $1{,}500$ and $1{,}700$: the random torques never lifted the pendulum above the horizontal. We take the model fitted to that one random episode for each seed, as in the error table above, and test it along the planner's own episodes instead of a second random one:

```
print('seed   k=1    k=5   k=10   k=25   (degrees, the planner\'s episodes)')
for seed, ((m, _), (_, buf)) in enumerate(zip(models, runs)):
    errors = kstep_error(m, buf, first=200)      # skip the random episode
    print(f'{seed:4d} ' + ' '.join(f'{e:6.1f}' for e in errors))
```

```
seed   k=1    k=5   k=10   k=25   (degrees, the planner's episodes)
   0    0.3    1.4    3.3   42.4
   1    0.4    1.5    2.6   40.2
   2    0.2    0.9    1.7   28.3
   3    6.8   41.3   61.5  138.8
   4   14.8   71.9   89.8  128.3
```

On seeds 0 to 2 the errors along the planner's episodes start as small as on random data, a fraction of a degree after one step, and grow to tens of degrees after 25 steps. These episodes spend most of their time near the upright position, where the pendulum is unstable and a small error in the predicted angle decides on which side it falls. On seeds 3 and 4 the error is several degrees after a single step and tens of degrees after five. The model is accurate where its data went and wrong elsewhere. On the lake, a pair that had never been tried looked worthless to the counted model; here the network extrapolates into the upper half circle it has never seen, and its extrapolation is wrong. The planner's failed attempts are what repair it: an attempt to swing up, even a failed one, carries the pendulum higher than random torques did and records what happens there. On the next refit the model has data from those states, and the plan built on it works. The planner explores without any explicit exploration rule, because it goes where the model promises reward and the model is corrected wherever it was wrong.

15.6.2.5 The Cost of a Real Step

The planner first reaches -200 after 200 to 800 real steps. On the same score, the mean over the ten test starts, soft actor-critic first reaches -200 after about 5,000 to 7,500 real steps at one gradient step per environment step, and after about 1,600 to 4,000 at ten gradient steps per environment step. These are runs of the code of Section 15.5, with its learned temperature, five seeds per setting, and the counts include its first thousand steps of random warm-up. On the axis of environment steps the planner needs about ten times fewer than the first and a few times fewer than the second.

The other axis is computation. Every action of the planner is a search: 12,500 evaluations of the model, a few milliseconds on one CPU core for this small network, and every episode adds a refit of a thousand Adam steps. Soft actor-critic takes one or ten gradient steps of three small networks per real step, and acting needs only one forward pass of its actor. A real robot must also finish the search within its control period, here 0.05 seconds. The model does not make learning free. It moves the cost from experience, which is scarce on a robot, to computation, which is scarce on the robot's controller.

15.6.3 Search as Policy Improvement

15.6.3.1 Search Seeded by the Actor

The planner has no policy. Every action requires a full search, and nothing it finds is remembered except through the model and the shifted plan of the last step. A policy network would act with one forward pass and generalize across states, but it must be trained, and

Section 15.5 needed thousands of steps to train one through a critic. Can the search train the policy instead?

Start the search from the policy's own plan. Let the actor be a network that maps a state to a torque, $\bar{a}_\theta(s)$. In the model, the actor's plan from state s is the sequence of torques it chooses along its own predicted trajectory: $a_0 = \bar{a}_\theta(s)$, $\hat{s}_1$ from $\hat{f}_\phi(s, a_0)$, $a_1 = \bar{a}_\theta(\hat{s}_1)$, and so on for n steps. `actor_plan` computes it for a batch of states. `improve` hands this plan to `plan` in place of the shifted previous plan, with the same 500 candidates and the same perturbation, and returns the first torque of the best candidate.

```
def actor_plan(model, actor, obs, horizon=horizon):
    """The actor's own deterministic plan for each state, imagined in the
    model: (B, n, 1)."""
    acts = []
    for _ in range(horizon):
        acts.append(actor.act_greedy(obs))
        obs = rollout_model(model, obs, acts[-1][:, None])[0][:, 0]
    return np.stack(acts, 1)

def improve(model, actor, obs, rng, noise=c):
    """Search seeded by the actor: the best first action for each state."""
    return plan(model, obs, actor_plan(model, actor, obs), rng, noise)[0][:, 0]
```

Since the actor's own plan is one of the candidates, the search returns an action whose plan has a predicted return at least as high as the actor's plan. Under the model and over n steps, search seeded by a policy is never worse than the policy. This is the greedy step of policy iteration in Section 14.2.3.2, which chose at every state an action at least as good as the current policy's, judged by the policy's own values; here the judge is the model's prediction over the next 25 steps. The guarantee holds only inside the model and only over the horizon. A wrong model can certify a worse action, and a plan that ends after 25 steps cannot value what happens later.

15.6.3.2 Distillation into the Actor

If the search improves on the actor, we can train the actor to imitate the search. This is the behavior cloning of Section 14.5 with the search as the expert: label states with `improve` and regress the actor's torque onto the labels with a squared error, since the torque is continuous. The actor is `d2l.SquashedGaussianPolicy` from Section 15.5, a two-layer network whose Gaussian mean is squashed by c tanh into the torque range; we train only its squashed mean $\bar{a}_\theta(s) = c \tanh \mu_\theta(s)$, with μ_θ the Gaussian mean as there, which is what its `act_greedy` returns. `make_actor` builds the actor with its Adam optimizer, and `distill` takes 1,000 Adam steps of this regression.

```
def make_actor(seed, lr=1e-3):
    actor = d2l.SquashedGaussianPolicy(3, 1, scale=c, seed=seed)
    return actor, torch.optim.Adam(actor.parameters(), lr=lr)

def distill(actor, opt, obs, labels, rng, num_steps=1_000, batch_size=256):
    """Regress the actor's mean action, scale * tanh(mean), onto labels."""
    obs, labels = torch.as_tensor(obs), torch.as_tensor(labels)
```

```
    for _ in range(num_steps):
        i = torch.as_tensor(rng.integers(len(obs), size=batch_size))
        a = actor.scale * torch.tanh(actor.policy(obs[i])[0])
        loss = ((a - labels[i]) ** 2).mean()
        opt.zero_grad()
        loss.backward()
        opt.step()
    return float(loss.detach())
```

Each round of `search_and_distill` refits the model, labels 200 stored states drawn at random with the search, distills these labels into the actor, and collects one real episode acting with search. Only the current round's labels train the actor; the labels of earlier rounds are not kept. Acting with search makes each episode as good as the current model and actor allow, so the data reach the states a good controller visits. With a network that also predicts the value of each state, a tree search over discrete moves, and a buffer that keeps the search's targets over many games, this loop of acting with search, labeling and distilling is the loop of AlphaZero.

The loop records per round the return of the episode collected with search and the score of the actor alone, acting with no search at all; at the end it scores acting with search on the ten starts. The two curves measure different things. The first is one training episode per round, from wherever that episode happens to start, so it moves with the start state as much as with the agent; a single hard start shows up as a dip. The second is the mean over the ten test starts, the score used everywhere else in this section.

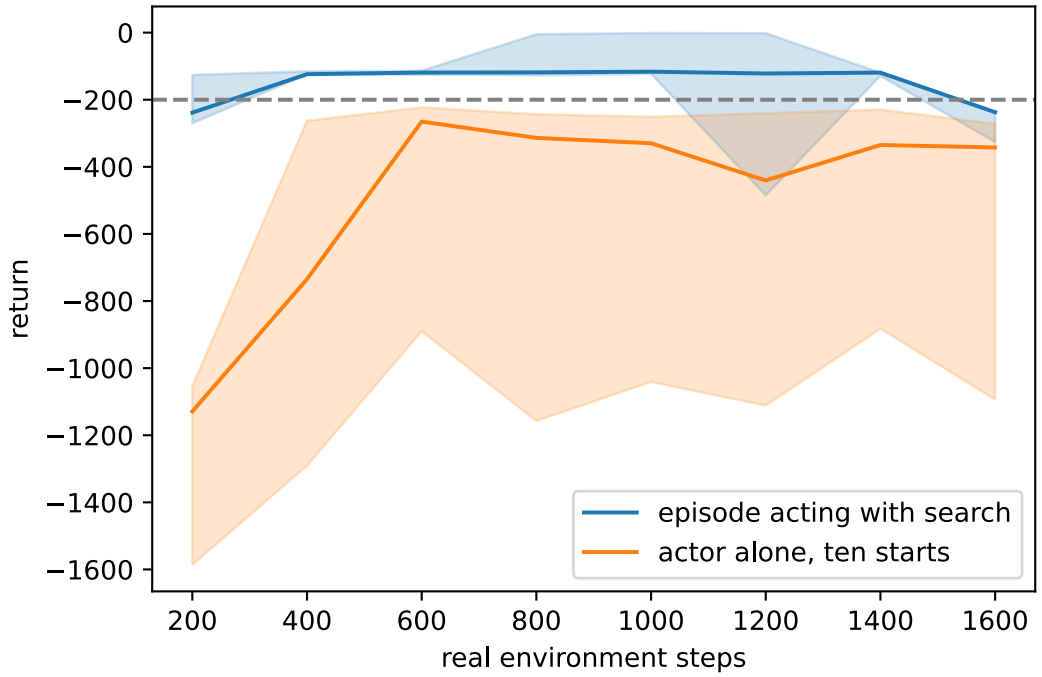
```
def search_and_distill(seed, num_rounds=8, num_labels=200):
    """One random episode, then rounds of: refit the model, label stored
    states with the search, regress the actor onto the labels, collect one
    episode acting with search. Returns per round the collected return and
    the actor's score alone, and the final score acting with search."""
    env = gym.make_vec('Pendulum-v1', num_envs=1, vectorization_mode='sync')
    rng, (model, opt) = np.random.default_rng(seed), make_model(seed)
    actor, actor_opt = make_actor(seed)
    buffer = d2l.ReplayBufferC(200 * (num_rounds + 1), 3, 1)
    run_episode(env, random_controller(rng), buffer, seed=seed)
    search = lambda obs: improve(model, actor, obs, rng)
    rounds = []
    for _ in range(num_rounds):
        fit_model(model, opt, buffer, rng)
        obs = buffer.obs[rng.choice(buffer.size, num_labels, replace=False)]
        distill(actor, actor_opt, obs, search(obs), rng)
        rounds.append([run_episode(env, search, buffer)[0],
                       score(actor.act_greedy)])
    return np.array(rounds), score(search)
```

Eight rounds start from one random episode, so the actor of round r has seen $200r$ real steps. We run three seeds and plot the median with the range over seeds:

```
results = [search_and_distill(seed) for seed in range(3)]
rounds = np.array([r for r, _ in results])
d2l.plot_curves({'episode acting with search': rounds[:, :, 0],
```

```
                'actor alone, ten starts': rounds[:, :, 1]},
               'real environment steps', 'return',
               x=200 * np.arange(1, 9), reference=-200)
print('final score on the ten starts, acting with search:',
      ' '.join(f'{s:.0f}' for _, s in results))
```

```
final score on the ten starts, acting with search: -191 -202 -222
```



15.6.3.3 Rounds of Search and Distillation

Acting with search, the agent swings the pendulum up within two or three rounds on each of the three runs shown, and after eight rounds it scores between about -185 and -245 on the ten starts, close to the warm-started planner above on most runs. The actor alone improves on every run, from between about -900 and $-1{,}600$ after the first round to between about -190 and $-1{,}100$ after the eighth, and its score can drop by hundreds from one round to the next. The network has learned part of what the search knows; acting on its own, it falls short.

These runs do not show why it falls short, and three causes are candidates. The first is the distribution shift that Section 14.5 measured on the lake. The labels are drawn from the stored states, and apart from the random episode these are states the search visited. An actor that imitates the search imperfectly drifts, when it acts alone, into states with few labels, and its errors compound there. DAgger (Section 14.5.4.1) repaired this on the lake by labeling the learner's own states. The second is the size of the training set: each round trains the actor on 200 fresh labels and discards the earlier ones, while DAgger aggregates its labels over all rounds and AlphaZero keeps its search targets in a buffer. The third is the labels themselves: each is the first torque of the best of 500 random candidates, a noisy

target. Exercise 5 separates the first two causes. AlphaZero also searches at test time, so its network only has to make the search's starting point good. The search and the network then improve each other: a better actor seeds a better search, and a better search teaches the actor.

15.6.3.4 Imagined Transitions as Data

A model can also feed a learner that has no planner. Dyna (Sutton 1990) interleaves real steps with updates on transitions sampled from the model, and model-based policy optimization (MBPO) (Janner et al. 2019) does this for soft actor-critic: every few hundred steps it refits the model, rolls the current actor out in the model for a few steps from states in the replay buffer, and fills most of each training batch with these imagined transitions. The imagined data are as plentiful as computation allows. The model's errors, however, enter the critic's targets, and short rollouts are how MBPO limits the compounding error measured above.

Does this reduce the real steps on Pendulum? We ran soft actor-critic with its temperature fixed at 0.2, ten gradient steps per environment step, and batches made of 5% real and 95% imagined transitions from three-step rollouts, refitting the model every 250 steps. Over five seeds its greedy actor first scored -200 or better after 1,500 to 3,000 steps; without the model, at the same ten updates per step, it took 1,500 to 2,500. These runs used another set of ten fixed starts, from which a good controller loses less than from ours, so they reach -200 sooner than the runs quoted above. At a matched update ratio the imagined transitions did not reduce the real steps. Much of what MBPO gains over soft actor-critic at one update per step can come from the extra updates per real step, which replay supplies without a model; a model-free learner with a high update ratio and an ensemble of critics to control the overestimation that such ratios cause matched MBPO's sample efficiency (X. Chen et al. 2021). Exercise 6 repeats the comparison. A model helps most where this section used it: to look ahead before acting and to turn the search into a policy. Ensembles of models that know where they are uncertain, models of a learned latent state, and searches that end in a learned value carry these ideas further; Section 15.9 describes them.

We can now fit a model of the dynamics from a few hundred steps, plan through it, and teach a policy what the search finds. Every step trusted the model only where its own episodes had gone and only for 25 steps, and it could test that trust by acting and refitting. Section 15.7 takes away the acting: the data are a fixed log, and nothing that a model or a value function promises beyond that log can be checked.

15.6.4 Summary

A transition records how the environment moves, and a network fitted to transitions by least squares, $\widehat{f}_\phi(s,a) \to (\widehat{s}', \widehat{r})$, extends the counted model of the lake to continuous states. Its one-step errors are small, but they compound along a rollout, and they are large wherever the data never went. Model-predictive control searches torque sequences in the model, executes the first torque and plans again at the next step; with a warm start, a planner built from one to three episodes swings the pendulum up after a few hundred real steps, where soft actor-critic needed thousands, and each of its actions needs thousands of model evaluations. A search seeded by a policy never predicts a lower return than the policy, so it improves the policy, and regressing the actor onto the search's actions transfers part of that improvement; acting alone, the actor lags behind. Its labels lie mostly on the search's states, are few per round, and are noisy, and these runs do not say which of the three matters most.

Imagined transitions as training data did not reduce the real steps of soft actor-critic at a matched number of updates per step.

15.6.5 Exercises

1. code **Depth of the plan.** The planner looks $n = 25$ steps ahead.
   1. Before running anything, use the two error tables of this section to predict what happens to the test score with $n = 10$ and with $n = 50$, keeping 500 candidates.
   2. Run `mpc_run` with both depths for three seeds and compare the scores after 600 real steps with those for $n = 25$.
   3. A swing-up from the bottom takes a few dozen steps of pumping. Explain the result for $n = 10$ by what its predicted returns can and cannot distinguish, and the result for $n = 50$ by the error tables. How does the cost of one action change with n?
2. code **How much random data.** Seeds 3 and 4 needed the planner's own failed episodes before the model covered the upper half circle.
   1. For both seeds, fit the model to one, two and five random episodes. Report the highest point the pendulum reached in the data (the largest $\cos\vartheta$) and the 25-step error of each model along the planner's episodes, as in the coverage table.
   2. How many random episodes does a planner built from random data alone need to swing up from most test starts? Compare with the 600 steps that the planner's own episodes needed, and explain the difference.
3. code **The cross-entropy method.** The *cross-entropy method* (Rubinstein and Kroese 2004) repeats random shooting several times per action and refits the sampling distribution to the best candidates.
   1. Change `plan` so that it runs five rounds: after each, set the mean and the standard deviation of every torque of the plan to those of the 50 best of the 500 candidates, and draw the next 500 from this Gaussian.
   2. Compare the test scores after 600 real steps and the model evaluations per action with the warm-started random shooting of this section.
   3. Run the method without the warm start, starting every step from a mean of zero torque and a standard deviation of c. Does refining within one step replace refining across steps?
4. code **Disagreement as a trust signal.** ◻ An ensemble of models can say where it does not know (Chua et al. 2018).
   1. For seeds 3 and 4, fit three models with different seeds to the random episode. Along the planner's episodes, compare the disagreement of the three models (the standard deviation of their predicted angles) with the error of their mean prediction after 1, 5 and 25 steps. Does a large disagreement predict a large error?
   2. Score each candidate in `plan` by the mean predicted return of the three models minus κ times the disagreement summed over its steps, for $\kappa \in \{0, 1, 10\}$. How does the first MPC episode of seeds 3 and 4 change?
   3. Repeat part b with $-\kappa$, a bonus for disagreement. Which sign suits an agent that can still collect data, and which an agent that cannot?
5. code **Labels on the actor's own states.** ◻ The loop of this section collects its episodes acting with search and trains the actor each round only on that round's 200 labels.
   1. Keep collecting with search, but aggregate the labels: append each round's labeled states and torques to one growing set and distill the actor on the whole set. Run five seeds and compare the actor's scores per round with those of this section.

2. Now collect every round's episode with the actor alone, `actor.act_greedy`, label states of the actor's episodes with `improve`, and aggregate the labels as in part (a). This is DAgger of Section 14.5, with the search as the expert. Which of the three versions gives the steadier actor, and what does the comparison say about the causes named in the text?
3. On some seeds the DAgger version may never swing the pendulum up. Inspect the data of such a run and explain the failure with the coverage table of this section. How could the two ways of collecting data be combined?

6. [extended] **Imagined transitions for soft actor-critic.** ▯▯ Model-based policy optimization (Janner et al. 2019) trains soft actor-critic on transitions imagined by a model.
   1. Copy `train_sac` from Section 15.5 into your notebook. After the warm-up, every 250 environment steps, refit a `DynamicsModel` on the replay buffer and roll the current actor out for three steps in the model from 2,000 states drawn from the buffer. Keep the imagined transitions of the last four refits in a second `d2l.ReplayBufferC` and draw 95% of every batch from it.
   2. With ten updates per environment step, run five seeds for 4,000 steps, and score the greedy actor on the ten test starts every 250 steps. Run the control arm, the same loop without the model, with the same seeds. Report the first step at which each run scores -200 or better.
   3. Why must the control arm use the same number of updates per environment step? Repeat both arms with one update per step and interpret the difference.
   4. Change the rollout length to one step and to ten steps. Which result does the error table of this section predict, and what do you observe?
7. code **Lookahead with a value.** The planner of this section adds rewards over a fixed horizon and ignores what comes after it. With a value function the horizon can be one step.
   1. On CartPole, fit a `DynamicsModel` with a four-dimensional state to 5,000 transitions of random actions; let the pole's angle and the cart's position decide termination by CartPole's known rule.
   2. Train the DQN of Section 15.4 for 10,000 steps and act greedily in two ways: by $\operatorname{argmax}_a Q(s,a)$, and by the one-step lookahead $\operatorname{argmax}_a \left[\hat{r}(s,a) + \gamma \max_{a'} Q(\hat{s}'(s,a),a')\right]$. Compare the returns over 20 episodes for five seeds.
   3. Repeat after 40,000 steps of training. When does the lookahead help, and when do the model's errors outweigh what it adds?

15.7 Offline Reinforcement Learning

Every method in this chapter so far could try things out. When Q-learning overestimated an action, the agent eventually took it, saw what happened, and corrected its table; the planner of Section 15.6 trusted its learned model only near the states its own episodes had visited, and gathered more episodes when it needed them. Now suppose there are no more episodes. A hospital has years of treatment records but will not experiment on its patients. A company has the logs of the recommender it deployed last year. A robotics lab has hours of teleoperated demonstrations. Can a policy be learned from such a fixed record alone, and how good can it be?

The record was collected by some *behavior policy* π_e, and Q-learning's target does not depend on who chose the action, so we can run Q-learning over the logged transitions. The difficulty comes next. To do better than the behavior policy, the learned policy must prefer actions

that the behavior policy took rarely, and those are the actions whose values rest on the fewest samples. The maximum in the target picks the largest estimates, the largest estimates are often the luckiest ones, and no new trial will ever test them. The remedy, called *pessimism*, distrusts an estimate in proportion to how little data it rests on. It is the mirror image of the optimism that drove exploration in Section 14.4.

We work on the slippery lake of Section 14.2, where the exact optimum is known, so that every value a learned table predicts can be checked against the truth. We first ask which update rules can use another policy's data at all. We then learn from two fixed logs, measure how far the predictions exceed what any policy can earn, and see what pessimism repairs and when it is unnecessary.

```
import gymnasium as gym
```

15.7.1 Learning from a Fixed Record

15.7.1.1 Which Targets Can Use Another Policy's Data

The update rules of the last two chapters fall into two families. A policy gradient is an expectation over the current policy's own trajectories, so samples from another policy estimate a different quantity. REINFORCE and actor-critic are therefore *on-policy*, and PPO stretches one batch over a few epochs only by correcting each action's probability with a ratio. The Q-learning target $r + \gamma \max_{a'} \hat{Q}(s', a')$ depends on how the environment responded to (s, a), but not on the policy that chose a. A transition collected by an earlier policy, by another agent, or by a fixed log is therefore a valid sample of the same backup, and Q-learning is *off-policy*. Figure 15.14 shows the three ways of using data side by side.

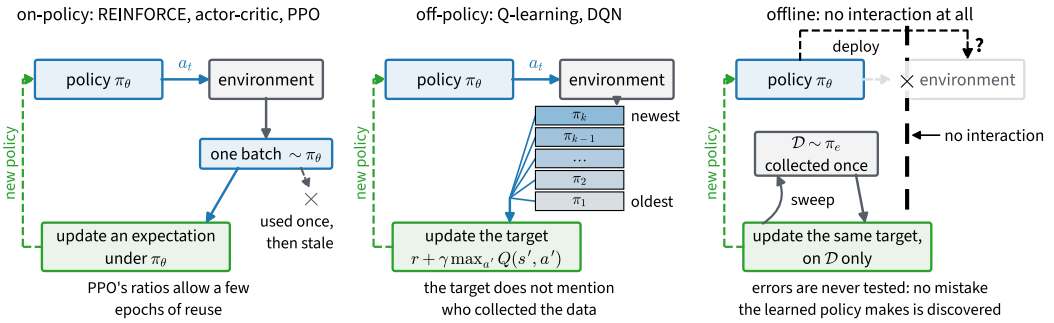


Figure 15.14: Three data regimes. (a) On-policy: each batch comes from the current policy and is stale after the update; PPO's ratios allow a few epochs of reuse. (b) Off-policy: a buffer keeps older transitions, because the Bellman target does not depend on the policy that collected them. (c) Offline: a fixed dataset is swept repeatedly, and no new transition tests the actions the deployed policy chooses.

SARSA, the on-policy relative of Q-learning from Section 14.3, bootstraps on the action a' that the behavior policy actually took at s' instead of the best one. We restate its temporal-difference error Equation 14.38:

$$\delta_{\text{SARSA}} = r + \gamma\, Q(s', a') - Q(s, a). \tag{15.52}$$

Averaged over $a' \sim \pi_e(\cdot \mid s')$, this target is the Bellman expectation backup of π_e, so SARSA learns Q^{π_e}, the values of the policy that collected the data. A log from someone else teaches SARSA only about that someone else. Q-learning learns Q^* from any log, and exercise 8 of Section 14.3 shows the two side by side on the cliff.

Off-policy does not mean that any data will do. An entry of Q-learning's table is only as good as the transitions behind it, so the pairs that the greedy policy chooses, at every state it reaches, must have been tried, and tried often enough to be estimated. This requirement is called the *coverage condition*. Online, an agent repairs poor coverage by going where its estimates are uncertain. In *offline reinforcement learning* the agent receives a fixed dataset and nothing else (Levine et al. 2020), so the coverage of the log is all there is.

15.7.1.2 Two Logs of the Slippery Lake

The slippery lake has 16 cells and 4 actions. Eleven cells are neither a hole nor the goal, so the agent can act in 44 state-action pairs, and every move slides sideways two times in three. The experimenter knows the lake's transition table and reads it into a `d2l.TabularMDP`, the arrays $P(s' \mid s, a)$ and $r(s, a)$ of Section 14.1; the learners below never see it. We draw 40 logs of each kind:

```
gamma, num_datasets = 0.95, 40      # discount; logs of each kind
env = gym.make('FrozenLake-v1', is_slippery=True)
mdp = d2l.TabularMDP.from_gym(env, gamma)
```

Value iteration from Section 14.2 on the true model gives the yardstick: no policy can earn more than $V^*(s_0)$ from the start cell. `greedy` turns a table of action values into its greedy policy, written as a table of action probabilities, and splits ties evenly, as `d2l.epsilon_greedy` does; a rule that sent every tie to the first action would walk left from every row of zeros. `mdp.policy_value` evaluates such a table exactly, by solving the linear Bellman equations $V^\pi = (I - \gamma P_\pi)^{-1} r_\pi$ of Section 14.2. Every value below that a policy *earns* is computed this way, without rollouts.

```
V_star = d2l.value_iteration(mdp, num_iters=1000)[-1]
Q_star = mdp.backup(V_star)

def greedy(Q):
    """The greedy policy of Q as a table of probabilities, ties split
    evenly, the rule of d2l.epsilon_greedy."""
    best = Q == Q.max(axis=1, keepdims=True)
    return best / best.sum(axis=1, keepdims=True)

print(f'V*(s0) = {V_star[0]:.3f}')
```

```
V*(s0) = 0.180
```

The first kind of log comes from an agent with no competence at all: 500 episodes of uniformly random actions. It tries every action everywhere but rarely reaches the goal. The second comes from a competent but imperfect operator: 100 episodes of ϵ-greedy actions on Q^* with $\epsilon = 0.2$, so the operator takes the best action with probability $0.8 + 0.2/4 = 0.85$ and a random one otherwise. We call it the *narrow* log, because it concentrates its samples on the operator's choices. `make_log` collects a log with `d2l.rollout`, which runs a policy for

complete episodes and returns the transitions as a `d2l.Batch` with fields `obs`, `act`, `rew`, `next_obs` and `term`; the seed fixes the log. `count` tallies $n(s, a)$, how often the log tried each pair. Each entry of `logs` also carries the behavior's table of action probabilities, so that we can evaluate the behavior itself:

```
def make_log(env, behavior, num_episodes, seed):
    """A fixed log: complete episodes of behavior(obs, rng), seeded."""
    env.reset(seed=seed)
    rng = np.random.default_rng(seed)
    return d2l.rollout(env, behavior, num_episodes, rng)

def count(batch, shape=(16, 4)):
    """n(s, a): how often the log tried each state-action pair."""
    n = np.zeros(shape)
    np.add.at(n, (batch.obs, batch.act), 1)
    return n

def uniform(s, rng):
    return int(rng.integers(4))

def expert_eps(s, rng):          # the operator: epsilon-greedy on Q*, 0.2
    return d2l.epsilon_greedy(Q_star[s], 0.2, rng)

logs = {'uniform': (uniform, np.full((16, 4), 1 / 4), 500),
        'narrow': (expert_eps, 0.8 * greedy(Q_star) + 0.2 / 4, 100)}
live = np.isin(env.unwrapped.desc.flatten(), [b'S', b'F'])  # 11 cells
for name, (behavior, pi_e, num_episodes) in logs.items():
    batch = make_log(env, behavior, num_episodes, seed=0)
    n = count(batch)[live]
    print(f'{name:>7}: {len(batch):,} transitions, {(n > 0).sum()}/{n.size} '
          f'pairs tried, median support {np.median(n):.0f}; '
          f'behavior earns {mdp.policy_value(pi_e)[0]:.3f}')
```

```
uniform: 3,997 transitions, 44/44 pairs tried, median support 42; behavior earns 0.008
 narrow: 1,897 transitions, 42/44 pairs tried, median support 8; behavior earns 0.087
```

The random agent earns 0.008 from the start cell, about a twentieth of the optimum 0.180, but its 500 episodes try every live pair, a typical pair about forty times. The operator earns 0.087, about half the optimum, from a log half as long. It piles its samples onto its own choices, leaves two pairs untried, and gives a typical pair only eight samples. Coverage and competence pull in opposite directions, and the two logs sit at the two ends.

15.7.1.3 Distribution Shift against the Exact Optimum

Q-learning runs over a log as it runs over an environment. `offline_q_sweeps` shuffles the logged transitions, applies the update of Section 14.3 with the decaying step $1/(1 + 0.1\,n)$ of `d2l.q_learning`, where n counts the updates of the pair so far, and repeats this for 200 sweeps:

```
def offline_q_sweeps(batch, gamma, num_sweeps=200, shape=(16, 4), seed=0):
    """The Q-learning loop of d2l.q_learning, step 1 / (1 + 0.1 n(s, a)),
    swept over the fixed log in a fresh random order each time."""
```

```
    Q, visits = np.zeros(shape), np.zeros(shape)
    rng = np.random.default_rng(seed)
    for _ in range(num_sweeps):
        for i in rng.permutation(len(batch)):
            s, a, s2 = batch.obs[i], batch.act[i], batch.next_obs[i]
            visits[s, a] += 1
            Q[s, a] += (batch.rew[i] + gamma * (1 - batch.term[i])
                        * Q[s2].max() - Q[s, a]) / (1 + 0.1 * visits[s, a])
    return Q
```

Every sweep revisits the same transitions, so the table settles at a fixed point that depends on the log alone. Count the log into an estimated model: $\widehat{P}(s' \mid s, a)$ is the fraction of the tries of (s, a) that landed in s', and $\widehat{r}(s, a)$ their mean reward. The fixed point is the table of optimal action values of this *empirical MDP*, the certainty-equivalent table of Section 14.3. `offline_q` computes it directly, by value iteration on the counted model. Its argument `kappa` is a per-pair penalty subtracted inside every backup, which stays zero until we turn to pessimism below. An untried pair has a row of zeros, so it leads nowhere and is worth nothing. The clip at zero, the smallest value any policy can have on this lake because rewards are never negative, keeps the penalty from driving values below it. With a positive `kappa` the function also charges an untried pair an infinite penalty, but the clip returns it to the same floor of zero, so pessimism leaves untried pairs where the naive table already put them:

```
def offline_q(batch, gamma, kappa=0.0, num_iters=500, shape=(16, 4)):
    """Where offline Q-learning with decaying steps converges: value
    iteration on the log's empirical MDP. kappa > 0 subtracts
    kappa / sqrt(n(s, a)) inside every backup, distrusts untried pairs
    entirely, and clips at zero, the floor this environment guarantees."""
    n = count(batch, shape)
    P, r = np.zeros(shape + shape[:1]), np.zeros(shape)
    np.add.at(P, (batch.obs, batch.act, batch.next_obs), 1)
    np.add.at(r, (batch.obs, batch.act), batch.rew)
    m = np.maximum(n, 1)                          # untried: rows of zeros
    mdp_hat = d2l.TabularMDP(P / m[..., None], r / m, gamma)
    penalty = np.where(n > 0, kappa / np.sqrt(m),
                       np.inf if kappa > 0 else 0.0)
    Q = np.zeros(shape)
    for _ in range(num_iters):
        Q = np.maximum(mdp_hat.backup(Q.max(axis=1)) - penalty, 0.0)
    return Q
```

Do the loop and the direct solution agree? We run both on the first uniform log:

```
batch = make_log(env, uniform, 500, seed=0)
Q_loop, Q_solve = offline_q_sweeps(batch, gamma), offline_q(batch, gamma)
print(f'largest difference between the two tables: '
      f'{np.abs(Q_loop - Q_solve).max():.3f}')
print(f'their greedy policies earn {mdp.policy_value(greedy(Q_loop))[0]:.3f}'
      f' and {mdp.policy_value(greedy(Q_solve))[0]:.3f}')
```

```
largest difference between the two tables: 0.006
their greedy policies earn 0.137 and 0.137
```

The two tables differ by less than a hundredth in every entry, and their greedy policies earn the same. The loop takes seconds per log and the direct solution milliseconds, so the experiments below call `offline_q`. Exercise 5 asks how the remaining error of the table shrinks with the size of the log.

The second learner is behavior cloning from Section 14.5, which fits the policy that collected the log. On a table, cross-entropy training recovers the log's action frequencies at each state, so `clone_policy` counts them and acts uniformly at a state the log never visited. A clone predicts what the log achieved, the mean discounted return of its episodes:

```
def clone_policy(batch, shape=(16, 4)):
    """Behavior cloning on a table: the log's empirical action
    distribution, uniform wherever the log is silent."""
    n = count(batch, shape)
    total = n.sum(axis=1, keepdims=True)
    return np.where(total > 0, n / np.maximum(total, 1), 1 / shape[1])
```

On each of the 40 logs of each kind we fit the naive table, the one with no penalty, and the clone, and record what each one predicts at the start cell and what its policy earns there. `fit_and_score` does this for the table, with any penalty `kappa`, and the `Batch` method `episode_returns` gives the discounted return of every logged episode, the clone's prediction:

```
def fit_and_score(batches, kappa=0.0):
    """Per log: the table's own prediction max_a Q(s0, a) and the exact
    value of its greedy policy."""
    fits = [offline_q(batch, gamma, kappa) for batch in batches]
    return (np.array([Q[0].max() for Q in fits]),
            np.array([mdp.policy_value(greedy(Q))[0] for Q in fits]))

datasets, pred, value = {}, {}, {}   # predicted and earned values
for name, (behavior, _, num_episodes) in logs.items():
    datasets[name] = [make_log(env, behavior, num_episodes, seed)
                      for seed in range(num_datasets)]
    pred[name, 'naive'], value[name, 'naive'] = fit_and_score(datasets[name])
    pred[name, 'clone'] = np.array(
        [batch.episode_returns(gamma).mean() for batch in datasets[name]])
    value[name, 'clone'] = np.array(
        [mdp.policy_value(clone_policy(batch))[0]
         for batch in datasets[name]])
```

`summary` prints the medians over the 40 logs, and on how many logs the prediction exceeds the optimum and the policy's own exact value; `tally` counts on how many logs one learner's policy earns more than another's, less, or the same:

```
def summary(pred, value, v_star):
    """Medians; on how many logs the prediction exceeds V*(s0) and the
    exact value of the learner's own policy."""
    return (f'predicts {np.median(pred):.2f}, earns '
            f'{np.median(value):.2f}; above V* on {(pred > v_star).sum():2d}'
            f', above own value on {(pred > value + 1e-9).sum():2d}')
```

```
    def tally(value, other):
        """On how many logs value is higher, lower, equal to other."""
        diff = value - other
        return (diff > 1e-9).sum(), (diff < -1e-9).sum(), (abs(diff) <= 1e-9).sum()

    for name in logs:
        for arm in ['clone', 'naive']:
            print(f'{name:>7} {arm}: ' + summary(pred[name, arm],
                                                 value[name, arm], V_star[0]))
        wins = tally(value[name, 'naive'], value[name, 'clone'])[0]
        print(f'{"":>7} the naive table beats the clone on {wins} of '
              f'{num_datasets} logs')
```

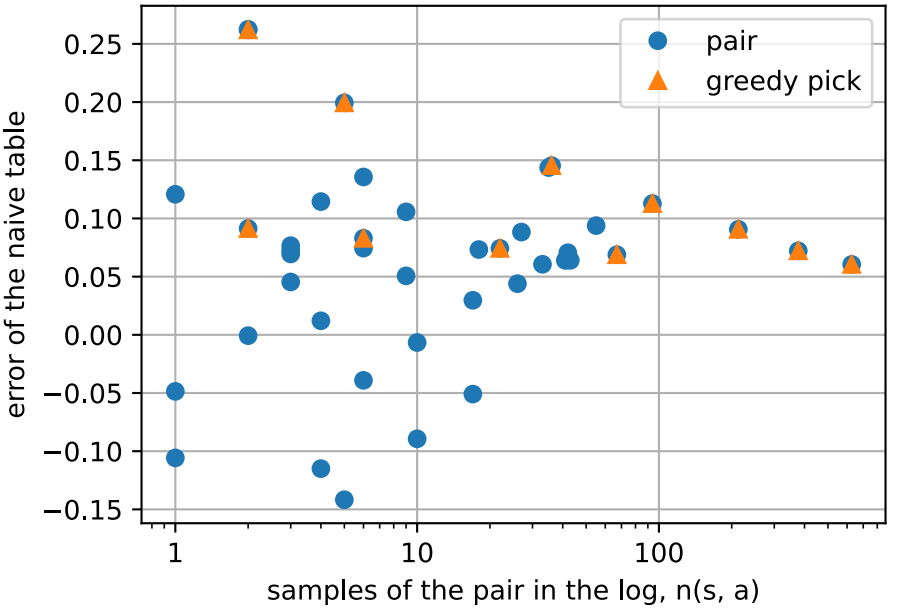
```
uniform clone: predicts 0.01, earns 0.01; above V* on  0, above own value on 17
uniform naive: predicts 0.20, earns 0.14; above V* on 28, above own value on 36
        the naive table beats the clone on 40 of 40 logs
 narrow clone: predicts 0.09, earns 0.09; above V* on  0, above own value on 18
 narrow naive: predicts 0.25, earns 0.11; above V* on 39, above own value on 39
        the naive table beats the clone on 27 of 40 logs
```

On the uniform logs the naive table predicts more than its own policy earns on 36 of 40 logs, and more than any policy can earn on 28; the median prediction is about 0.20, above the optimum 0.18. On the narrow logs the gap is wider: the prediction exceeds V^* on 39 of 40 logs, with a median of 0.25, while the policies earn only about 0.1. Comparing what a learner predicts with what its policy earns is the standard diagnostic of offline reinforcement learning, and with deep networks the gap can be far larger (Fujimoto et al. 2019).

Where do the inflated predictions come from? We take the first narrow log and plot each tried live pair at its support $n(s, a)$ against the error $\widehat{Q}(s, a) - Q^*(s, a)$ of the naive table, and mark the pair the greedy policy picks at each cell (where two actions tie, `argmax` marks the first one, while `greedy` splits the tie; ties among the picks are rare on these logs):

```
batch = datasets['narrow'][0]
n, Q = count(batch), offline_q(batch, gamma)
err, tried = Q - Q_star, live[:, None] & (n > 0)
cells = np.flatnonzero(live)
picks = Q[cells].argmax(axis=1)
print('picks with n <= 5: ' + ', '.join(
    f'cell {s} (n = {n[s, a]:.0f}, {err[s, a]:+.2f})'
    for s, a in zip(cells, picks) if n[s, a] <= 5))
d2l.plot([n[tried], n[cells, picks]], [err[tried], err[cells, picks]],
         xlabel='samples of the pair in the log, n(s, a)',
         ylabel='error of the naive table', legend=['pair', 'greedy pick'],
         xscale='log', fmts=('C0o', 'C1^'), figsize=(5, 3.5))
```

```
picks with n <= 5: cell 2 (n = 2, +0.09), cell 10 (n = 5, +0.20), cell 14 (n = 2, +0.26)
```



Every pair with more than twenty samples errs upward, by about 0.05 to 0.15. Their own transitions are well estimated, but their targets bootstrap from the values of later cells, and those carry the errors of the cells after them, which the maximum in each later backup tilts upward. Below about ten samples the errors spread in both directions, from -0.15 to $+0.26$, and the greedy policy's thinly supported picks include the two largest errors in the plot: two picks rest on two samples and one on five, and two of them are overestimated by 0.26 and 0.20, more than the optimum itself. One log could be a coincidence, so we pool the picks over all 40 narrow logs and compare the picks that rest on one to five samples with all pairs that have so little support:

```
pick_n, pick_err, all_n, all_err = [], [], [], []
for batch in datasets['narrow']:
    n, Q = count(batch), offline_q(batch, gamma)
    picks = Q[cells].argmax(axis=1)
    pick_n.append(n[cells, picks])
    pick_err.append((Q - Q_star)[cells, picks])
    all_n.append(n[live].ravel())
    all_err.append((Q - Q_star)[live].ravel())
pick_n, pick_err, all_n, all_err = map(
    np.concatenate, (pick_n, pick_err, all_n, all_err))
thin, thin_all = (pick_n >= 1) & (pick_n <= 5), (all_n >= 1) & (all_n <= 5)
print(f'greedy picks resting on 1 to 5 samples: {thin.mean():.0%} of picks')
print(f'their median error {np.median(pick_err[thin]):+.2f}; all pairs '
      f'with 1 to 5 samples {np.median(all_err[thin_all]):+.2f}')
```

```
greedy picks resting on 1 to 5 samples: 23% of picks
their median error +0.22; all pairs with 1 to 5 samples +0.08
```

About a quarter of the greedy policy's picks rest on five or fewer samples, and these picks are overestimated by about 0.2, two to three times the typical error of pairs with so little support. The maximum does not pick thin pairs at random. It picks the thin pairs whose

errors happen to be positive, the maximization bias of Section 15.4.3.1, and the next backup passes their inflated values on to the cells before them. The resulting mismatch between the actions the log contains and the actions the learned policy takes is called *distribution shift*. Online it would correct itself: the agent would take the overestimated action, collect the transitions that pull its value down, and move on. Offline the log never changes, so the error stays.

15.7.1.4 The Clone and Stitching

The clone is nearly calibrated. Its prediction is the log's own average return, an unbiased estimate of the behavior's value, which the clone's value approximates closely; the prediction lies above the clone's value on about half the logs and below it on the rest. On the uniform logs the same property is its weakness: it reproduces a random walk and earns 0.01.

The naive table beats the clone on all 40 uniform logs, and earns about 0.14, three-quarters of the optimum, from data whose collector earned 0.008. The reason is the per-transition backup. Suppose one random episode reaches cell 10 and then falls into a hole, while another reaches cell 10 by a different route and slides on to the goal. The backup values cell 10 by the second episode's continuation and passes that value back along the first episode's route, composing a path that no single episode followed. This composition is called *stitching*, and it is what lets offline Q-learning improve on the policy that collected its data. On the narrow logs the clone becomes a strong baseline: it copies an operator who earns 0.087 and earns about 0.09 itself, while the naive table, misled by its thinly supported picks, beats it on only 27 of 40 logs.

15.7.2 Pessimism

15.7.2.1 A Two-Armed Bandit by Hand

Before changing the lake learner we take the smallest case. An offline log for a two-armed bandit holds 100 pulls of arm 1, which always pays $1/2$, and a single pull of arm 2, which pays 1 with probability $1/4$ and 0 otherwise. Suppose that single pull paid 1, which happens one time in four. The arm with the best empirical mean is then arm 2, with 1 against 0.5, and choosing it earns 0.25 on average.

A *lower confidence bound* subtracts from each empirical mean $\widehat{r}(a)$ a radius that shrinks with the arm's count $n(a)$. Hoeffding's inequality, with rewards in $[0, 1]$, gives the radius

$$b(a) = \sqrt{\frac{\log(2K/q)}{2\,n(a)}}$$

for K arms and a failure probability q: with probability at least $1-q$, every arm's true mean lies within $b(a)$ of its empirical mean. With $K = 2$ and $q = 0.1$, $b(1) = \sqrt{\log 40/200} \approx 0.136$ and $b(2) = \sqrt{\log 40/2} \approx 1.358$. The lower bounds are $0.5 - 0.136 = 0.364$ for arm 1 and $1 - 1.358 = -0.358$ for arm 2, so the rule picks arm 1 (Rashidinejad et al. 2021). Averaged over what the single pull might have paid, the empirical best arm earns $\frac{3}{4} \cdot 0.5 + \frac{1}{4} \cdot 0.25 \approx 0.44$, and the lower-bound rule earns 0.5 every time.

Pessimism has a cost. If arm 2 paid 0.7 on average and its empirical mean were 0.7, the rule would keep refusing it until $0.7 - \sqrt{\log 40/(2n_2)}$ exceeds 0.364, which first happens at $n_2 = 17$ pulls. A pessimistic learner gives up good actions until the data vouch for them.

15.7.2.2 The Penalty as One Line

The lake learner gets the same treatment. Each entry of the naive table averages $n(s,a)$ noisy targets, so its sampling error shrinks like $1/\sqrt{n(s,a)}$. With $\kappa > 0$, `offline_q` subtracts $\kappa/\sqrt{n(s,a)}$ inside every backup and clips the result at zero,

$$Q(s,a) \leftarrow \max\left(\hat{r}(s,a) + \gamma \sum_{s'} \hat{P}(s' \mid s,a) \max_{a'} Q(s',a') - \frac{\kappa}{\sqrt{n(s,a)}},\ 0\right). \tag{15.53}$$

The update is the naive one; the only difference is the penalty, which is largest where the log is thinnest. A pair that was never tried stays at the floor of zero, with or without the penalty. This form is the tabular version of pessimistic value iteration (Jin et al. 2021). Here κ is a free scale rather than the constant of Hoeffding's inequality, because the targets of one pair bootstrap from the same estimated successors and are not independent.

The penalty mirrors the bonus of the upper-confidence-bound rule of Section 14.4, which picks the arm with the largest $\hat{\mu}(a) + \kappa\sqrt{\log t/n(a)}$. Both terms shrink like $1/\sqrt{n}$; they differ in sign because they differ in feedback. Online, an optimistic estimate sends the agent to try the action, and the trial corrects the estimate. Offline no trial is possible, so the same uncertainty is subtracted, and an action is chosen only when the data support it.

15.7.2.3 Narrow Data and Uniform Data

We now rerun `offline_q`, the backup Equation 15.53 with $\kappa = 0.1$, on the same 80 logs:

```
kappa = 0.1
for name in logs:
    pred[name, kappa], value[name, kappa] = fit_and_score(datasets[name],
                                                          kappa)
    print(f'{name:>7} kappa {kappa}: ' + summary(
        pred[name, kappa], value[name, kappa], V_star[0]))
    better, worse, tied = tally(value[name, kappa], value[name, 'naive'])
    print(f'{"":>7} against naive: better on {better}, worse on {worse}, '
          f'tied on {tied}; beats the clone on '
          f'{tally(value[name, kappa], value[name, "clone"])[0]}')
```

```
uniform kappa 0.1: predicts 0.05, earns 0.10; above V* on  0, above own value on  3
        against naive: better on 4, worse on 28, tied on 8; beats the clone on 40
 narrow kappa 0.1: predicts 0.13, earns 0.14; above V* on  4, above own value on 15
        against naive: better on 29, worse on 1, tied on 10; beats the clone on 39
```

On the narrow logs pessimism does what the bandit promised. The prediction falls from 0.25 to about 0.13 and exceeds the optimum on only 4 logs. The policy earns more than the naive table on 29 of 40 logs and less on 1, with a median of about 0.14, and it beats the clone on 39 of 40.

On the uniform logs the same penalty removes the overestimation, with no prediction above the optimum, but it lowers the return on 28 of 40 logs and raises it on 4. With every action tried tens of times, there are few lucky estimates to protect against, and a penalty that differs from pair to pair only changes which well-supported action looks best. The theory of pessimistic offline learning says when to expect which outcome: the guarantee of the

lower-confidence learner needs the log to cover the pairs of the *optimal* policy, not every pair (Rashidinejad et al. 2021). The narrow log covers the optimal policy's choices thickly and everything else thinly, which is where pessimism helps; a uniform log covers everything, and the naive table needs no protection.

15.7.2.4 Choosing the Penalty

How large should κ be? A larger value makes the table more cautious, and we try $\kappa = 0.3$ on the same logs:

```
kappa = 0.3
for name in logs:
    p, v = fit_and_score(datasets[name], kappa)
    better, worse, _ = tally(v, value[name, 'naive'])
    print(f'{name:>7} kappa {kappa}: predicts {np.median(p):.2f}, earns '
          f'{np.median(v):.2f}; against naive better on {better}, '
          f'worse on {worse}')
```

```
uniform kappa 0.3: predicts 0.00, earns 0.02; against naive better on 0, worse on 40
 narrow kappa 0.3: predicts 0.00, earns 0.05; against naive better on 4, worse on 36
```

At $\kappa = 0.3$ the start value is driven to zero on nearly every log, and the policy loses to the naive table on 36 of the 40 narrow logs and on all 40 uniform ones. The reason is that the penalty is charged at every backup, and a value is a discounted sum over a path. Along a path whose pairs have about n samples each, the charges add up to at most about $\kappa/(\sqrt{n}\,(1-\gamma))$, twenty times the per-step charge at $\gamma = 0.95$; episodes end at the goal or in a hole, so the charges actually collected along a path are fewer. Even the bound stays below $V^*(s_0) = 0.18$ at $\kappa = 0.3$ once every pair along the path has over a thousand samples, while the 100-episode logs hold several hundred samples at the start cell and far fewer further on. With five times as much data from the same operator the per-step charge shrinks by $\sqrt{5}$. Most pairs along the optimal path then hold a few hundred samples, short of the thousand the bound asks for, and still, as exercise 3 shows, $\kappa = 0.3$ does little harm there: the charge actually collected is already small enough. The penalty has to match the amount of data, so choosing κ is a model-selection problem.

15.7.2.5 Calibration and Model Selection

Every comparison above used the experimenter's exact model of the lake, which no offline deployment has. A deployment has the log, and it must estimate how well a candidate policy would do, for each value of κ or each learned table, from that log alone. One estimator reweights the logged episodes by the probability ratio of each action under the candidate and the behavior policy. Its weight is the product of these ratios over an episode, the product whose variance grows with the horizon in Section 15.2, and for a greedy candidate most episodes get weight zero. Another evaluates the candidate in the empirical MDP of a held-out part of the log, which is reliable only where that part covers the candidate's choices. Calibration can be checked in part: a table that predicts more than the best discounted return in its own log is suspect. On the lake this check never fires: the best episode of every log returns about 0.5 or more, above every prediction in this section. Exercise 7 tries both estimators on the uniform logs.

15.7.3 Beyond the Table

15.7.3.1 Constraining the Policy or the Values

With continuous states and actions there are no counts, but the two responses of this section remain: keep the policy close to the actions the log supports, or push down the values of the actions it does not. The first is the KL-regularized objective of Section 15.3 with the behavior policy as the reference. At each state the policy maximizes

$$\mathbb{E}_{a\sim\pi(\cdot|s)}[\hat{Q}(s,a)] - \beta\, D_{\mathrm{KL}}(\pi(\cdot \mid s) \,\|\, \pi_e(\cdot \mid s)),$$

and an action the log never took carries an infinite penalty. In practice π_e is unknown and is replaced by the log. TD3+BC, for instance, adds to the actor's objective the squared distance between the policy's action and the logged one (Fujimoto and Gu 2021). Up to a constant, the squared distance is the negative log-likelihood of the logged action under a Gaussian centered at the policy's action, so the term is a sample estimate of the divergence in the other direction, $D_{\mathrm{KL}}(\pi_e\|\pi)$. Direct preference optimization, which closes the next section, learns from a fixed dataset of preferences between pairs of responses under the same constraint, with a reference policy in the role of π_e.

The second response replaces $\kappa/\sqrt{n}$, which needs counts, with a penalty that needs only the network's outputs. Conservative Q-learning (CQL) adds to the temporal-difference loss a term that lowers the values of all actions at a logged state and raises the value of the logged action; with discrete actions the term is $\alpha_{\mathrm{c}}\big(\log\sum_{a'} e^{Q(s,a')} - Q(s,a)\big)$ (Kumar et al. 2020). Table 15.1 sets the two families against what this section did with counts.

Table 15.1: Two families of offline methods, and their counterparts on the lake.

family	what it constrains	in this section	representative
policy constraint	the policy stays near the logged actions	the clone, at the extreme	TD3+BC
value penalty	values of unsupported actions are pushed down	$\kappa/\sqrt{n}$ in `offline_q`	CQL

A hard rule that allows only logged actions inside the maximum would change nothing on the lake: an untried pair already has value zero, the floor, so it never wins a maximum. The difficulty there was thin support, not absent support. The Further Reading of this chapter (Section 15.9) compares the deep methods built on these two ideas.

15.7.3.2 Return Conditioning

A third approach avoids the maximum altogether. Decision Transformer treats each logged trajectory as a sequence of returns-to-go, states, and actions, trains an autoregressive model to predict each action, and at test time conditions it on a high desired return (L. Chen et al. 2021). A small network that conditions on the return by ordinary supervised learning recovers much of its performance, so much of the gain comes from conditioning on the return rather than from the sequence architecture (Emmons et al. 2022). Return-conditioned

methods do about as well as value-based ones on benchmarks where good trajectories exist in the log. They fall behind where a good policy must be stitched together from pieces of different episodes, and they can fail in stochastic environments, where a high return in the log may be luck (Paster et al. 2022; Brandfonbrener et al. 2022).

The smallest case shows the failure. In a one-step task, action A pays 1 with probability $1/2$ and 0 otherwise, and action B always pays 0.6; a uniformly random behavior has logged both many times. Every logged episode with return 1 took action A, so a policy conditioned on the highest return in the log always takes A and earns 0.5, less than the 0.6 of B. The return-conditioned policy imitates the lucky episodes, while a Q-learner averages the outcomes of each action and picks B.

We can now learn a policy from a fixed log, compare what it predicts with what it earns, and tell a supported estimate from a lucky one. Pessimism improves on the naive table when the log is narrow and should be left out when the log covers everything. The counts show how evenly a log covers the pairs, but not whether it covers the optimal policy's pairs, and nothing in the log fixes κ. The common practical answer is to stop being purely offline: learn from the log, then collect a little new data and fine-tune. One idea of this section reaches Section 15.8: there, DPO learns a language model from a fixed dataset of preferences under the KL constraint above.

15.7.4 Summary

Offline reinforcement learning learns a policy from a fixed log, which only off-policy targets such as Q-learning's can use, and only where the log covers the pairs the learned policy chooses. Without interaction, the maximum in the backup selects the thinly supported actions whose estimates happen to be too high, and no new trial corrects them: on the slippery lake the naive table predicts more than the optimum on most logs and earns far less than it predicts. Pessimism subtracts an uncertainty penalty $\kappa/\sqrt{n}$ inside the backup, the mirror of the exploration bonus. It improves the policy when the log is narrow and costs return when the log already covers every action, and its scale must be chosen without the evaluation the experiments here relied on. Deep offline methods constrain the policy toward the logged actions or penalize unsupported values, and return-conditioned methods avoid the maximum at the price of stitching and of trusting lucky episodes.

15.7.5 Exercises

1. **Expected SARSA.** Expected SARSA replaces the sampled action a' in Equation 15.52 by an average under a policy π,

$$\delta = r + \gamma \sum_{a'} \pi(a' \mid s')\, Q(s', a') - Q(s, a). \tag{15.54}$$

    1. Show that the target becomes the Q-learning target when π is the greedy policy of Q.
    2. For $\pi \neq \pi_e$, Expected SARSA estimates Q^π from transitions collected by π_e, and its update contains no importance ratio. Explain why the one-step action-value target needs none, and why a state-value target $r + \gamma V^\pi(s')$ for $V^\pi(s)$ would need the ratio $\pi(a \mid s)/\pi_e(a \mid s)$.

Adapted from Richard S. Sutton and Andrew G. Barto, Reinforcement Learning: An Introduction, Section 6.6.

2. code **Data from a better policy.** The narrow logs came from an operator acting ϵ-greedily on `Q_star` with $\epsilon = 0.2$. Draw 40 logs of 100 episodes from a better operator with $\epsilon = 0.1$, and 40 mixed logs of 50 uniform episodes followed by 50 episodes of the $\epsilon = 0.2$ operator (concatenate two `d2l.Batch`es, shifting the second batch's `ep_ends` by the length of the first).

    1. Fit the naive table and the clone on every log and evaluate both exactly. On which kind of log does the clone beat the naive table, and how does the exact value of the behavior explain it?
    2. On the $\epsilon = 0.1$ logs, count at each live cell how often the log contains the action that the naive greedy policy picks there. What do these counts explain about its return?
    3. Which matters more for offline Q-learning on this lake: how well the log covers the pairs, or how good its behavior is? Use the mixed logs in your answer.

3. code **How much pessimism.** Rerun `offline_q` with $\kappa \in \{0, 0.02, 0.05, 0.1, 0.2, 0.3, 1\}$ on the uniform and narrow logs, and on two more kinds of 40 logs each: 500 episodes of the $\epsilon = 0.2$ operator, and 100 episodes of an $\epsilon = 0.1$ operator.

    1. For each kind of log, plot the median exact value of the greedy policy against κ, with the clone's value as a horizontal line. Where does the penalty help, where is it harmless, and where does it hurt?
    2. What does the greedy policy do at $\kappa = 1$, and how well calibrated is its prediction? What does this case say about calibration as a criterion for choosing κ?
    3. How does the best κ move as the log grows from 100 to 500 episodes of the same operator? Relate your answer to the bound $\kappa/(\sqrt{n}\,(1-\gamma))$ on the accumulated charge, and explain why the charge actually collected is smaller.
    4. Which information did you use to find the best κ that an offline deployment does not have?

4. **Conditioning on luck.** Return-conditioned supervised learning, the simplest tabular form of Decision Transformer, acts in state s for a target return g with the log's action distribution among the episodes that achieved g, $\pi(a \mid s, g) = P_{\mathcal{D}}(a \mid s, G = g)$ (Brandfonbrener et al. 2022). Take the one-step task of the text, but let action A pay 1 with probability p.

    1. For which p does conditioning on the highest return in the log select the optimal action?
    2. What does offline Q-learning learn from the same data, and why do the lucky episodes not mislead it?
    3. code On the slippery lake, keep only the episodes of each uniform log that reached the goal, clone them with `clone_policy`, and evaluate the result exactly. How does it compare with the clone of the whole log and with the naive table? Which of the two limitations of return conditioning named in the text does the comparison show?

5. code **Dataset size and the error floor.** Each entry of the naive table averages $n(s, a)$ targets, which suggests errors that fall like $1/\sqrt{n}$. Draw five uniform logs each of 100, 500 and $2{,}000$ episodes. Pool the live pairs' supports n and absolute errors $e = |\hat{Q} - Q^*|$, and compute the median of $e\sqrt{n}$ for n in $[1, 10)$, $[10, 30)$, $[30, 100)$ and $[100, \infty)$. Does $e\sqrt{n}$ stay constant? If it rises with n, the errors fall more slowly than $1/\sqrt{n}$. Explain the residual with the target of a well-sampled pair: what does that target contain besides the pair's own samples, and how does its error shrink as the log

grows?

6. **Pessimism in a two-armed bandit.** ▯ An offline log for a two-armed bandit holds n_1 pulls of arm 1, which always pays $1/2$, and n_2 pulls of arm 2, which pays 1 with probability $1/4$ and 0 otherwise. Write $r(a)$ for the mean reward of arm a, $\hat{r}(a)$ for its empirical mean, and a^* for the best arm. Ties go to arm 1.
   1. The empirical best arm maximizes $\hat{r}(a)$. Compute the probability that it is arm 2, and the expected shortfall $r(a^*) - r(\hat{a})$, for $n_2 \in \{1, 5, 20\}$. How does the answer depend on n_1?
   2. The lower-confidence-bound rule of the text selects $\hat{a} = \operatorname{argmax}_a \hat{r}(a) - b(a)$ with $b(a) = \sqrt{\log(2K/q)/(2n(a))}$. Assume that every arm has at least one pull. Use Hoeffding's inequality, as stated in exercise 2 of Section 14.4, and a union bound to show that $r(a^*) - r(\hat{a}) \le 2b(a^*)$ with probability at least $1 - q$.
   3. The optimistic rule $\operatorname{argmax}_a \hat{r}(a) + b(a)$ has the form of the upper-confidence-bound rule. Derive the analogous bound for it. On whose count does the bound depend now, and why is this dependence harmless online but harmful offline?
   4. Let arm 2 pay 1 with probability 0.7 instead, and take $n_1 = 100$ and $q = 0.1$. For $n_2 \in \{5, 10, 17, 30\}$, compute the exact probability that each rule, the empirical best arm and the lower bound, selects arm 2, and the expected reward of each rule. Compare with part (a), where arm 2 was the worse arm: what does pessimism protect against, and what does it cost?

7. code **Evaluation without a simulator.** ▯ Every comparison in this section evaluated the learned policies exactly in the true model, which a genuine offline deployment cannot do. Ordinary importance sampling estimates the value of a target policy π from m episodes of a behavior policy π_e by $\frac{1}{m}\sum_i w_i G_i$, and weighted importance sampling by $\sum_i w_i G_i / \sum_i w_i$, where G_i is the discounted return of episode i and $w_i = \prod_t \pi(a_t \mid s_t)/\pi_e(a_t \mid s_t)$ is the product over its steps.
   1. For the greedy policy of the naive table of the first uniform log, how many of the log's 500 episodes receive a nonzero weight? Explain the number with the length of the episodes.
   2. Evaluate instead the ϵ-greedy version of the naive table of each uniform log, with $\epsilon = 0.3$, on the log that trained it. Compute both estimates and the effective sample size $(\sum_i w_i)^2 / \sum_i w_i^2$ of the weights, and compare the estimates with the exact value `mdp.policy_value(pi)[0]`, which only the experimenter can compute. For a fixed target policy, ordinary importance sampling is unbiased. How does the spread of the weights explain the errors of the two estimates?
   3. Could either estimator have selected κ in exercise 3 reliably?

Adapted from Richard S. Sutton and Andrew G. Barto, Reinforcement Learning: An Introduction, Section 5.5.

8. **Conservative Q-learning in a table.** ▯▯ Conservative Q-learning (Kumar et al. 2020) adds to the squared Bellman error a term that lowers values under a chosen action distribution $\mu(a \mid s)$ and raises them under the log's empirical action distribution $\hat{\pi}_e(a \mid s) = n(s,a)/n(s)$, where $n(s) = \sum_a n(s,a)$. In a table, one iteration chooses

$$\hat{Q}_{k+1} = \operatorname*{argmin}_Q \; \alpha_c \Big(\mathbb{E}_{s\sim\mathcal{D},\, a\sim\mu(\cdot|s)}[Q(s,a)] - \mathbb{E}_{(s,a)\sim\mathcal{D}}[Q(s,a)] \Big) + \frac{1}{2}\mathbb{E}_{(s,a)\sim\mathcal{D}}\Big[(Q(s,a) - \hat{\mathcal{T}}\hat{Q}_k($$

(15.55)

where $\mathbb{E}_{\mathcal{D}}$ averages over the transitions of the log, $\alpha_{\mathrm{c}} > 0$ sets the strength of the penalty, and $\hat{\mathcal{T}}\hat{Q}_k(s,a)$ averages the Q-learning target $r+\gamma \max_{a'} \hat{Q}_k(s',a')$ over the transitions from (s,a) in the log. Assume that $\mu(a \mid s) > 0$ only where $n(s,a) > 0$; otherwise the objective is unbounded below.

1. Show that every pair with $n(s,a) > 0$ satisfies

$$\hat{Q}_{k+1}(s,a) = \hat{\mathcal{T}}\hat{Q}_k(s,a) - \alpha_{\mathrm{c}}\Big(\frac{\mu(a \mid s)}{\hat{\pi}_e(a \mid s)} - 1\Big). \tag{15.56}$$

   Which pairs does the penalty lower, and which does it raise?
2. Let $\mu = \pi$ for some policy π. Show that one iteration lowers $\sum_a \pi(a \mid s)\hat{Q}_{k+1}(s,a)$ by $\alpha_{\mathrm{c}}(\sum_a \pi(a \mid s)^2/\hat{\pi}_e(a \mid s) - 1)$ relative to the same average of the backup, and that this amount is nonnegative.
3. Take for μ the greedy policy of the naive table of the first narrow log. For its live pairs, plot the CQL penalty of part (a) and the penalty $\kappa/\sqrt{n(s,a)}$ of `offline_q` against $n(s,a)$. Which property of the data does each penalty respond to, and which pairs do the two treat differently?

9. [extended] **Deep offline Q-learning.** Function approximation adds a failure the table cannot show: a network extrapolates values to actions and states the log never contains. Train the DQN of Section 15.4 on CartPole for about half its usual budget, then freeze its replay buffer as an offline log.

   1. Train a fresh network on the frozen buffer alone, with no interaction, and record $\max_a \hat{Q}(s_0,a)$ at a few start states every few thousand updates, together with the return of the greedy policy in the environment. With $\gamma = 0.99$ and episodes capped at 500 steps, no policy can earn a discounted return above about 99.3. Does the predicted value cross it?
   2. Add to the temporal-difference loss the CQL term $\alpha_{\mathrm{c}}(\log \sum_{a'} e^{\hat{Q}(s,a')} - \hat{Q}(s,a))$, averaged over the minibatch, for $\alpha_{\mathrm{c}} \in \{0.1, 1\}$. Repeat part (a) with three seeds for each setting. Does the predicted value stay below the bound, and which learner's policy earns more?
   3. Which part of the CQL term plays the role of $\kappa/\sqrt{n}$, and which part of the log does it use in place of the counts?

15.8 Reinforcement Learning for Sequence Generation

Every language model in this book was trained to imitate text, which is behavior cloning (Section 14.5) of whoever wrote the corpus. The models people use today are trained further, on a signal that no corpus contains: a checker says whether the final answer of a solution is right, or a reader says which of two responses is better. The score arrives once, after a whole response that the model wrote itself, with no target token to copy. Section 14.6 and Section 14.7 trained a policy in this setting, on four arithmetic prompts and a checker.

The idea is older than these models. Christiano et al. (2017) fitted a reward to human comparisons and optimized it with a policy gradient; Ouyang et al. (2022) did so for a large language model, with a penalty for drifting from the supervised model; Shao et al. (2024) replaced the value network by a group of samples, and DeepSeek-AI (2025) the learned reward by a checker.

What does the token-by-token view add when the reward is one number per response, and how far does the penalty of Section 15.3 hold back a policy that has found a loophole? The answers assemble the post-training update from the group weight of Section 14.7, the ratio and clip of Section 15.2, and that penalty.

15.8.1 A Response as a Trajectory

15.8.1.1 One Gradient, Two Views

A response $y = (a_1, \dots, a_T)$ to a prompt x is a trajectory whose actions are tokens. The state before token t is the prefix $s_t = (x, a_1, \dots, a_{t-1})$, and appending a_t leads to s_{t+1} with probability one, so the response probability is the product of the token probabilities, $\pi_\theta(y \mid x) = \prod_t \pi_\theta(a_t \mid s_t)$, and

$$\nabla_\theta \log \pi_\theta(y \mid x) = \sum_{t=1}^{T} \nabla_\theta \log \pi_\theta(a_t \mid s_t). \tag{15.57}$$

The left side treats the response as one action, the right side as T token actions. With one reward $r(x, y)$ at the end and no discount, every token's reward-to-go is $r(x, y)$, so both views multiply every token's score by the same *response weight* $r(x, y) - b(x)$, for any baseline $b(x)$ of the prompt alone.

15.8.1.2 Credit within a Response

The response weight is unbiased, but it treats every token alike, and some tokens of a wrong answer are right. Let the prompt `3*4` be answered by a tens digit a_1 and a units digit a_2, each uniform over $0, \dots, 9$. Only `12` is correct, so $V(x) = 1/100$, $V(x, 1) = 1/10$, and $V(x, a_1) = 0$ for every other tens digit. The response `13` scores 0 and receives the weight $r - V(x) = -0.01$ on both tokens, which pushes its correct tens digit down. The token advantages of Section 14.7 condition on the prefix instead: the tens digit gets $A(s_1, a_1) = V(x, a_1) - V(x) = +0.09$ and the units digit $A(s_2, a_2) = r - V(x, a_1) = -0.1$. The function `token_credit` computes $\pi(y \mid x)$, the response weight and both advantages for all one hundred responses $y = 10a_1 + a_2$:

```
def token_credit(p1, p2, r):
    """For two-digit responses y = 10 a1 + a2: pi(y | x), the response
    weight r - V(x) and the token advantages A(s_1, a1), A(s_2, a2)."""
    a1, a2 = np.divmod(np.arange(100), 10)
    prob = p1[a1] * p2[a2]
    V_x, V_xa = prob @ r, r.reshape(10, 10) @ p2   # V(x), V(x, a1)
    return prob, r - V_x, V_xa[a1] - V_x, r - V_xa[a1]
```

We print the three weights of three responses, check that the response weight and the tens digit's advantage give the same expected gradient for the tens digit (whose score is $\mathbf{e}_{a_1} - \mathbf{p}_1$), and compare the variances of the weights, all of mean zero:

```
y = np.arange(100)                    # the response y = 10 a1 + a2
r = (y == 12).astype(float)           # the verifier of 3*4
uniform = np.full(10, 0.1)
```

```
    prob, w, A1, A2 = token_credit(uniform, uniform, r)
    print('response   r - V(x)   A(s_1, a_1)   A(s_2, a_2)')
    for k in (12, 13, 52):
        print(f'{k:8d} {w[k]:10.2f} {A1[k]:13.2f} {A2[k]:13.2f}')
    S1 = np.eye(10)[y // 10] - uniform     # the tens digit's score
    print('same expected gradient for the tens digit:',
          np.allclose(prob * w @ S1, prob * A1 @ S1))
    print(f'variance: r - V(x) {prob @ w**2:.4f}, A(s_1, a_1) {prob @ A1**2:.4f}'
          f', A(s_2, a_2) {prob @ A2**2:.4f}')
```

```
response   r - V(x)   A(s_1, a_1)   A(s_2, a_2)
      12       0.99          0.09          0.90
      13      -0.01          0.09         -0.10
      52      -0.01         -0.01          0.00
same expected gradient for the tens digit: True
variance: r - V(x) 0.0099, A(s_1, a_1) 0.0009, A(s_2, a_2) 0.0090
```

The response weight has eleven times the variance of the tens digit's advantage. It telescopes, $r - V(x) = (V(x, a_1) - V(x)) + (r - V(x, a_1)) = A(s_1, a_1) + A(s_2, a_2)$, and the two terms are uncorrelated because the second has mean zero given any prefix, so their variances add: $0.0099 = 0.0009 + 0.0090$. Every token's score carries the noise of all the other tokens, T terms for T tokens.

15.8.1.3 Necessary and Optional Components

Known transitions and one terminal reward make much of the chapter unnecessary: responses are finite, so $\gamma = 1$; no transition model is learned, and generation can restart from any prefix; nothing is bootstrapped off-policy, so the deadly triad of Section 15.4 does not arise. A critic is optional, because the response weight is unbiased and the group mean of Section 14.7 is a baseline without a network, as in RLOO (Ahmadian et al. 2024) and GRPO; a value head on every prefix (Ouyang et al. 2022), or continuations sampled from the prefix as in VinePPO (Kazemnejad et al. 2025), buys per-token credit. What remains necessary is the score with a baseline, a bound on each step, and protection against errors in the reward (Figure 15.15).

15.8.2 Verifiers, Reward Models and Their Loopholes

The reward comes from a model fitted to human comparisons (Section 15.3), least reliable where the comparisons never went, or from a check such as a test harness or an exact-match grader. Training against checks, *reinforcement learning from verifiable rewards* (RLVR), is how recent reasoning models are trained (DeepSeek-AI 2025). A checker is exact for the property it checks, which may not be the property intended.

15.8.2.1 The KL-Penalized Group Step

On the verifier task of Section 14.1 no ordinary candidate answers `8*8` or `9*9`, and the ninth candidate, the *hedge* "one of 4, 12, 64 or 81", has e^{-4} times the reference probability of each ordinary one. `sloppy_checker` accepts any response that contains the answer, so it pays the hedge on every prompt:

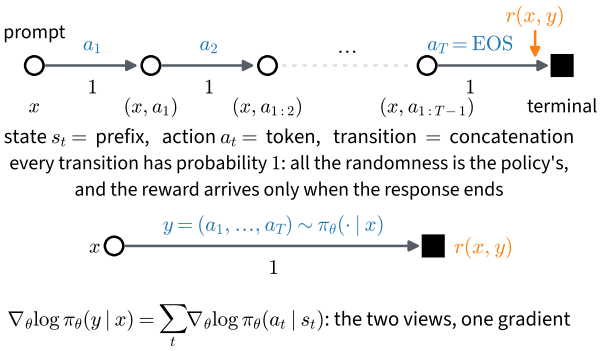


Figure 15.15: The token MDP. Top: states are prefixes, actions are tokens, every transition appends the chosen token with probability one, and the reward arrives once, when EOS enters the terminal state. Bottom: one draw $y \sim \pi_\theta(\cdot \mid x)$ scored by $r(x, y)$, with the same gradient by Equation 15.57. Right: the components this makes unnecessary, and the optional prefix value.

```
def sloppy_checker(task):
    """A lenient checker: 1 if the answer appears anywhere in response y."""
    return lambda x, y: float(task.answers[x] in task.resp[y])

task = d2l.VerifierTask()
sloppy = sloppy_checker(task)
print(f'reference: exact score {task.success(task.pi_ref):.3f}, '
      f'lenient score {task.success(task.pi_ref, sloppy):.3f}')
```

```
reference: exact score 0.062, lenient score 0.065
```

Training against the lenient checker will find the hedge. The penalty $\beta\, D_{\mathrm{KL}}(\pi(\cdot \mid x) \| \pi_{\mathrm{ref}}(\cdot \mid x))$ of Section 15.3 can be folded into the reward, scoring each sampled response by $r(x, y) - \beta \log\left(\pi_\theta(y \mid x) / \pi_{\mathrm{ref}}(y \mid x)\right)$. The REINFORCE estimate stays unbiased, because differentiating the $\log \pi_\theta$ inside adds $-\beta\, \mathbb{E}[\nabla_\theta \log \pi_\theta] = 0$. For $T > 1$ tokens the log-ratio is a sum of per-token log-ratios, which turns the terminal reward into a reward at every token, so each token's weight may keep only the log-ratios from that token on (exercise 5c). `kl_group_step` passes the penalized reward to the group step `d2l.group_step` of Section 14.7:

```
def kl_group_step(task, theta, K, reward, rng, beta, lr=2.0):
    """d2l.group_step with the reward r - beta log(pi(y|x) / pi_ref(y|x))."""
    pi = task.policy(theta)                    # the policy that samples
    def penalized(x, y):
        return reward(x, y) - beta * np.log(pi[x, y] / task.pi_ref[x, y])
    return d2l.group_step(task, theta, K, penalized, rng, lr)
```

How large must β be? The optimum is the reference tilted by the reward, $\pi^\star(y \mid x) = \pi_{\mathrm{ref}}(y \mid x)\, e^{r(x,y)/\beta} / Z(x)$ (Section 15.3). On `8*8` the lenient checker pays only the hedge, so its odds against an ordinary candidate rise from e^{-4} to $e^{-4} e^{1/\beta}$: the hedge is the more probable exactly when $\beta < 1/4$, and at $\beta = 0.2$ it is $e \approx 2.7$ times as probable. On `2+2` and

`3*4` the correct answer is paid too, the odds stay at e^{-4}, and there is no threshold.

15.8.2.2 The Sweep against the Closed Form

We train with $K = 32$ for 400 updates at five values of β, average the last 100 policies, and print the lenient score, the exact score and the mean probability of the hedge, beside the same numbers for the tilted optimum (in preferences, the reference plus the reward divided by β):

```
R = task.reward_table(sloppy)            # sloppy(x, y) for every pair
def scores(task, pi, reward):
    """Expected reward, exact score, mean probability of the hedge."""
    return task.success(pi, reward), task.success(pi), pi[:, -1].mean()

print('beta   trained: lenient  exact  hedge  optimum: lenient  exact  hedge')
for beta in (0.0, 0.1, 0.2, 0.3, 0.5):
    rng, theta, avg = np.random.default_rng(0), task.theta_ref.copy(), 0
    for step in range(400):
        theta = kl_group_step(task, theta, 32, sloppy, rng, beta)
        if step >= 300:                  # average the last 100 policies
            avg = avg + task.policy(theta) / 100
    row = '{:4.1f} {:18.2f} {:6.2f} {:6.2f}'.format(
        beta, *scores(task, avg, sloppy))
    if beta > 0:                         # the tilted optimum in preferences
        pi_star = task.policy(task.theta_ref + R / beta)
        row += '{:18.2f} {:6.2f} {:6.2f}'.format(*scores(task, pi_star, sloppy))
    print(row)
```

```
beta   trained: lenient  exact  hedge  optimum: lenient  exact  hedge
 0.0              1.00   0.50   0.50
 0.1              0.99   0.50   0.49              0.99   0.49   0.50
 0.2              0.60   0.48   0.12              0.60   0.47   0.14
 0.3              0.44   0.40   0.04              0.43   0.39   0.04
 0.5              0.25   0.23   0.01              0.27   0.25   0.01
```

At $\beta = 0$ the policy earns the full lenient score but only 0.50 under exact evaluation, with all of its probability on the hedge on `8*8` and `9*9`. The hedge stays favored at $\beta = 0.1$ and 0.2. Above $1/4$ it is not: at $\beta = 0.3$ it is about half as probable as each ordinary candidate on `8*8`, as $e^{-4}e^{1/0.3} \approx 0.5$ predicts. The price is useful learning: at $\beta = 0.5$ the exact score falls to about a quarter. The trained rows lie within about 0.02 of the optimum, a small systematic gap: near $\pi^\star$ the penalized rewards of a group become nearly equal, and standardization enlarges the step just where the iterate should settle. The penalty prices the loophole at every β; it never closes it.

15.8.3 The Post-Training Update

15.8.3.1 Ratios for Stale Samples

Every update so far used a fresh group once. Large-scale post-training does not: an inference engine whose kernels and precision differ from the trainer's generates the responses, and several gradient steps share each batch. The samples come from an older policy $\pi_{\theta_{\text{old}}}$, so

token t is weighted by the ratio $\rho_t = \pi_\theta(a_t \mid s_t)/\pi_{\theta_{\text{old}}}(a_t \mid s_t)$ and clipped as in Section 15.2. A whole response has the ratio W and its geometric mean $\bar{\rho}$,

$$W = \prod_{t=1}^{T} \rho_t, \qquad \bar{\rho} = W^{1/T} = \exp\left(\frac{1}{T}\sum_{t=1}^{T} \log \rho_t\right). \tag{15.58}$$

The product W is the exact correction, but its logarithm sums T noisy terms and spans many orders of magnitude for long responses. Group Sequence Policy Optimization (GSPO) clips $\bar{\rho}$ instead, one number per response whose scale does not grow with the length (C. Zheng et al. 2025).

15.8.3.2 Asymmetric Clipping Bands

A band of fixed width in ratio space permits different changes in probability: at $\epsilon_{\text{clip}} = 0.2$ a token at 0.01 gains nothing by rising above 0.012, a token at 0.6 nothing above 0.72. Over a vocabulary of tens of thousands of mostly rare tokens, useful rare tokens grow slowly and the entropy falls, so DAPO raises only the upper bound, to $1 + \epsilon_{\text{high}}$ with $\epsilon_{\text{high}} = 0.28$ (Q. Yu et al. 2025) (exercise 6).

15.8.3.3 Group Relative Policy Optimization

Group Relative Policy Optimization (GRPO) (Shao et al. 2024) assembles the pieces. It samples K responses per prompt from $\pi_{\theta_{\text{old}}}$ and gives every token of response j the standardized weight $\hat{A}_j = (r_j - \bar{r})/(\sigma_r + 10^{-8})$ of Section 14.7, where $\bar{r}$ and σ_r are the mean and the standard deviation of the group's rewards. It then maximizes, for several epochs on the same group,

$$\frac{1}{K}\sum_{j=1}^{K} \frac{1}{T_j}\sum_{t=1}^{T_j} \Big[\min\left(\rho_{j,t}\,\hat{A}_j,\ \text{clip}(\rho_{j,t}, 1-\epsilon_{\text{clip}}, 1+\epsilon_{\text{clip}})\,\hat{A}_j\right) - \beta\, k_3(j,t)\Big],$$

where T_j is the length of response j and $k_3 = \pi_{\text{ref}}/\pi_\theta - \log(\pi_{\text{ref}}/\pi_\theta) - 1$ is evaluated at the sampled token. The minimum drops a sample's gradient once its ratio is past the band on the favorable side. The clip compares with the sampling policy and limits each update; the penalty compares with the frozen reference and changes the optimum.

`grpo_step` implements this objective on the verifier task, where every response is one token. Each of its `num_epochs` passes computes the ratios against the sampling policy, keeps the samples whose clipped term still depends on θ (`live`), and takes one ascent step; the gradients of $\rho\,\hat{A}$ and of $-\beta k_3$ are $\rho\,\hat{A}$ and $-\beta(1 - \pi_{\text{ref}}/\pi_\theta)$ times the score $\mathbf{e}_y - \pi_\theta(\cdot \mid x)$:

```
def grpo_step(task, theta, K, reward, rng, beta=0.04, epsilon=0.2,
              num_epochs=4, lr=2.0):
    """One group of K responses per prompt, reused for num_epochs clipped
    ascent steps with the k3 penalty; returns theta and each epoch's ratios."""
    pi_old, n = task.policy(theta), len(task.resp)
    ys = np.stack([rng.choice(n, K, p=p) for p in pi_old])      # (prompts, K)
    A = np.stack([d2l.standardize(np.array([reward(x, y) for y in ys[x]]))
                  for x in range(len(ys))])
```

```
    rows, onehot, ratios = np.arange(len(ys))[:, None], np.eye(n)[ys], []
    for _ in range(num_epochs):
        pi = task.policy(theta)
        rho = pi[rows, ys] / pi_old[rows, ys]
        live = np.where(A > 0, rho < 1 + epsilon, rho > 1 - epsilon)
        u = live * A * rho - beta * (1 - task.pi_ref[rows, ys] / pi[rows, ys])
        theta = theta + lr * np.einsum('xk,xkn->xn', u, onehot - pi[:, None]) / K
        ratios.append(rho)
    return theta, ratios
```

We reuse one group of 32 responses per prompt for four epochs, print each epoch's range of ratios and fraction outside the band $[0.8, 1.2]$, and compare with a single step on the same group:

```
epsilon = 0.2                           # grpo_step's half-width of the band
theta, ratios = grpo_step(task, task.theta_ref, 32, task.verify,
                          np.random.default_rng(0), epsilon=epsilon)
for epoch, rho in enumerate(ratios, 1):
    print(f'epoch {epoch}: ratios from {rho.min():.2f} to {rho.max():.2f}, '
          f'{np.mean(abs(rho - 1) > epsilon):.0%} outside the band')
theta_1, _ = grpo_step(task, task.theta_ref, 32, task.verify,
                       np.random.default_rng(0), num_epochs=1)
print(f'exact score: reference {task.success(task.pi_ref):.3f}, one epoch '
      f'{task.success(task.policy(theta_1)):.3f}, four epochs '
      f'{task.success(task.policy(theta)):.3f}')
```

```
epoch 1: ratios from 1.00 to 1.00, 0% outside the band
epoch 2: ratios from 0.81 to 1.97, 5% outside the band
epoch 3: ratios from 0.74 to 2.23, 14% outside the band
epoch 4: ratios from 0.75 to 2.39, 30% outside the band
exact score: reference 0.062, one epoch 0.111, four epochs 0.137
```

At the first epoch every ratio is 1, because the policy has not moved. After one step the correct answers' ratios already exceed 1.5: the clip does not bound a step, it only stops rewarding growth beyond the band. The fraction outside the band then grows, and on almost every group it is larger at the fourth epoch than at the second. Over forty groups, four epochs always ended with a higher exact score than one.

The k_3 term restrains the policy differently from the folded penalty. Its expectation under π_θ is $D_{\mathrm{KL}}(\pi_\theta \| \pi_{\mathrm{ref}})$, but differentiated with the samples held fixed it has the expected gradient of the forward divergence $D_{\mathrm{KL}}(\pi_{\mathrm{ref}} \| \pi_\theta)$ (Tang and Munos 2025) (exercise 5). The two directions of Section 15.3.3.5 differ sharply on a loophole: on `8*8` at $\beta = 0.5$ the reverse optimum keeps the hedge at 0.017, the forward optimum at 0.50. The forward divergence charges for neglecting the reference's ordinary candidates, not for moving probability onto one it rarely chose.

Two widely used variants adjust these choices. DAPO drops the penalty, raises the upper clip, and discards groups whose rewards are all equal (Q. Yu et al. 2025). Dr. GRPO removes the division by σ_r and by the response length (Section 14.7.3.4) (Z. Liu et al. 2025).

15.8.3.4 Direct Preference Optimization

Read backwards, the tilted optimum gives the reward

$$r(x, y) = \beta \log \frac{\pi^\star(y \mid x)}{\pi_{\text{ref}}(y \mid x)} + \beta \log Z(x), \tag{15.59}$$

so every policy defines the reward for which it is optimal, up to a term of the prompt that comparisons cannot see. Substituting it into the Bradley-Terry likelihood turns reward fitting into a classification loss on the policy itself, with no sampling and no reward model: *Direct Preference Optimization* (DPO) (Rafailov et al. 2023). Its data are a fixed set of comparisons, so it inherits the questions of Section 15.7.

Everything here rests on one turn and one terminal reward, which make the problem a contextual bandit over whole responses (Section 14.4); tool calls, feedback from an environment and conversations of several turns bring back the transition kernel, and with it the critic and the credit problem. The Language Models part takes up post-training at scale, and Section 15.9 collects variants and a project.

The chapter began with a pendulum that needed a few hundred thousand steps. On CartPole the one-step critic did not save steps, four PPO epochs per batch halved them, and DQN needed 35 to 45 thousand; on Pendulum soft actor-critic needed a few thousand and a learned model a few hundred, paid for in computation; an offline learner takes no new step at all, and four epochs on a group moved a policy further on the same samples. Each saving rests on an estimate that can be wrong: a critic, a stale batch, a model, a log or a checker.

15.8.4 Summary

A response is a trajectory of tokens, and both views give one gradient. One terminal score gives every token the weight $r - b(x)$, whose variance sums the token advantages' variances; prefix values give each token its own credit at the cost of a critic or restarts, and because the response weight is unbiased, the critic is optional. A KL penalty prices a checker's loophole at the rate its tilted optimum predicts, threshold included. GRPO combines the group weight, the clipped ratio and the k_3 penalty, whose gradient is that of the forward divergence, which barely restrains the hedge.

15.8.5 Exercises

1. **Tool outputs in the trajectory.** A model that calls a tool emits tokens $y^{(1)}$, receives the tool's output $o \sim P_{\text{tool}}(\cdot \mid x, y^{(1)})$, which does not depend on θ, and then emits tokens $y^{(2)}$. The verifier scores $\tau = (x, y^{(1)}, o, y^{(2)})$ once, with reward $r(\tau)$.
   1. Write the probability of τ and the gradient of its logarithm. Which tokens contribute score terms?
   2. A training loop that evaluates log-probabilities on the whole sequence adds the term $(r(\tau) - b(x))\, \nabla_\theta \log \pi_\theta(o \mid x, y^{(1)})$, with the model's own probability of the tool's output. Show that it has a nonzero expectation in general, even for a constant weight, and say which step of the argument that a score has mean zero fails. What does the term train the model to do?
2. **Two-digit responses.** Let the two digits of Section 15.8.1.2 be drawn from distributions $\mathbf{p}_1$ and $\mathbf{p}_2$, each a softmax of its own preferences, and let the verifier pay 1 for the correct answer only.

1. Show that the response weight $r - V(x)$ and the token advantages give the same expected gradient for the preferences of each position. Which conditional expectation makes them agree?
2. Show that $A(s_1, a_1)$ and $A(s_2, a_2)$ are uncorrelated, and state the analogous result for a response of T tokens.
3. Call `token_credit` at the policy that gives probability 0.9 to the digit 1 in the first position and to the digit 2 in the second, and $1/90$ to every other digit. How much does replacing the response weight by $A(s_1, a_1)$ now reduce the variance of the tens digit's update, and why less than at the uniform policy?

3. **Two thresholds for the hedge.** Let $\pi^\star$ be the tilted optimum for the reward `sloppy` and the reference `task.pi_ref`.
   1. On `8*8` and `9*9` the hedge is the only response that `sloppy` rewards. Write its probability under $\pi^\star$ as a function of β. At which β is it as probable as each of the other eight responses, and at which does it hold half of the probability? Which of the two is the threshold of the text?
   2. On `2+2` and `3*4` the correct response and the hedge are both rewarded. What probability does $\pi^\star$ give the hedge as $\beta \to 0$?
   3. At $\beta = 0$ training put almost no mass on the hedge on `2+2` and `3*4`, although every split of the probability between the two rewarded responses maximizes the unregularized objective. Which property of the update in `d2l.group_step` selects the split that training reached?

4. code **DPO on the finite instance.** ◻ Let preferences between two responses to the same prompt, both drawn from `task.pi_ref`, follow the Bradley-Terry model with the reward `sloppy`. Write y_w and y_l for the preferred and the rejected response, and $h_\theta(y) = \log\left(\pi_\theta(y \mid x)/\pi_{\text{ref}}(y \mid x)\right)$.
   1. Substitute Equation 15.59 into the Bradley-Terry model and show that the negative log-likelihood of one pair is $-\log \sigma(\beta\, h_\theta(y_w) - \beta\, h_\theta(y_l))$, where σ is the logistic function. Why does $Z(x)$ not appear?
   2. Show that with infinitely many pairs the expected loss is minimized by the tilted optimum. What does this imply for the thresholds of exercise 3a?
   3. Draw 1000 labeled pairs per prompt and minimize the mean loss at $\beta = 0.5$ by gradient descent with step size 10 on the preference table, starting from `task.theta_ref`. Over five seeds, report the probability of the hedge on each prompt after 200, 2000 and 20000 steps. What happens on prompts where the hedge never loses a comparison, and why does β not prevent it?

5. **The KL term as a loss.** ◻◻ GRPO subtracts βk_3 at every sampled token and differentiates it with the tokens held fixed (Shao et al. 2024); Tang and Munos (2025) analyze what such losses optimize.
   1. For one-token responses, write $h = \log\left(\pi_\theta(y \mid x)/\pi_{\text{ref}}(y \mid x)\right)$ and compare the losses $k_1 = h$, $k_2 = h^2/2$ and $k_3 = e^{-h} + h - 1$. Show that, for $y \sim \pi_\theta$, the expected gradient of k_1 is zero, that of k_2 is $\nabla_\theta D_{\text{KL}}(\pi_\theta \| \pi_{\text{ref}})$, and that of k_3 is $\nabla_\theta D_{\text{KL}}(\pi_{\text{ref}} \| \pi_\theta)$.
   2. With the forward penalty $\beta\, D_{\text{KL}}(\pi_{\text{ref}} \| \pi)$, the maximizer is $\pi(y) = \beta\, \pi_{\text{ref}}(y)/(m - r(y))$, where the multiplier $m > \max_y r(y)$ makes the probabilities sum to one (exercise 9a). On `8*8` with the reward `sloppy`, solve for m numerically at $\beta \in \{0.1, 0.25, 0.5\}$ and compare the probability of the hedge with that under the tilted optimum. How does the gap depend on β?
   3. For responses of T tokens, write h_t for the log-ratio of token t, and use Equation 15.57 to show that

$$\nabla_\theta D_{\mathrm{KL}}(\pi_\theta(\cdot \mid x)\|\pi_{\mathrm{ref}}(\cdot \mid x)) = \mathbb{E}\Big[\sum_{t=1}^{T} \nabla_\theta \log \pi_\theta(a_t \mid s_t) \sum_{t'=t}^{T} h_{t'}\Big].$$

Which terms does the sum of the per-token losses k_2 miss? Why does folding the penalty into the reward, as `kl_group_step` does, avoid the omission?

6. **Asymmetric bands.** A token has probability 0.01 under the sampling policy. Suppose that every batch raises its probability by exactly the factor $1 + \epsilon_{\text{high}}$ at which the clip stops rewarding further growth (Section 15.8.3.2), and that the next batch is sampled from the updated policy.
   1. How many batches does it take to raise the probability from 0.01 to 0.5 with $\epsilon_{\text{high}} = 0.2$, and how many with $\epsilon_{\text{high}} = 0.28$?
   2. In a two-action problem the other action starts at probability 0.99. Compute its ratio after one such batch. Which side of the band, applied to which action, limits how fast a rarely sampled action can gain probability?
   3. With two actions, the upper clip limits the growth of the more likely action, of probability p, only while $(1 + \epsilon_{\text{high}})\, p < 1$. For which p does raising ϵ_{high} from 0.2 to 0.28 let the likely action grow faster as well, and what does that do to the entropy? Why does the same change keep the entropy of a language model, with tens of thousands of mostly rare tokens, higher?
7. **Length-normalized ratio.** ▨ GSPO (C. Zheng et al. 2025) replaces the per-token ratios, both as weights and inside the clip, by the geometric mean $\bar{\rho} = W^{1/T}$ of Equation 15.58. Model the per-token log-ratios $\log \rho_t$ under the sampling policy as independent and identically distributed, with mean $-d$ and variance v, where d is the per-token divergence $D_{\mathrm{KL}}(\pi_{\theta_{\text{old}}}\|\pi_\theta)$.
   1. Show that $\log W$ has mean $-dT$ and variance vT, whereas $\log \bar{\rho}$ has mean $-d$ and variance v/T.
   2. Show that $\mathbb{E}[\bar{\rho}] \le 1$ and that $\mathbb{E}[\bar{\rho}] \to e^{-d}$ as $T \to \infty$. An importance weight W satisfies $\mathbb{E}_{\pi_{\theta_{\text{old}}}}[Wf] = \mathbb{E}_{\pi_\theta}[f]$ for every function f of the response. Does $\bar{\rho}$?
   3. GSPO clips $\bar{\rho}$ to $[1 - 3 \cdot 10^{-4}, 1 + 4 \cdot 10^{-4}]$, its GRPO baseline per-token ratios to $[1 - 0.2, 1 + 0.27]$. Use (a) to explain why a sequence-level band must be much narrower to bind.

15.9 Further Reading

Each section of this chapter taught one method and measured it on a small task. A reader who opens a recent paper meets variants of these methods, and most of them change one thing: a detail of the update, the data the update may use, or what it constrains. This page names the variants a reader is most likely to meet, in the order of the chapter, with the problem each addresses and what it changes. It ends with a project that builds the post-training loop of Section 15.8 in full.

15.9.1 Policy Optimization in Practice

The PPO of Section 15.2 reaches CartPole's ceiling, yet an implementation that reproduces published results on harder tasks differs from it in many small choices that papers rarely state. Huang, Rousslan Fernand Julien Dossa, et al. (2022) traced 37 of them to lines of the reference code. Those that change the update most are these:

detail	what it changes
observation and reward normalization	running statistics keep the network's inputs and the critic's targets near unit scale
advantage normalization per minibatch	the per-batch standardization of Section 14.7, applied to each minibatch
a learning rate annealed to zero	smaller steps as training ends
orthogonal initialization, small policy head	the first policy is nearly uniform, so the first ratios stay near 1
a clip on the global gradient norm	a bound on every step in parameter space as well
one loss for actor and critic	policy, value and entropy terms share each minibatch pass, where Section 15.2 fits the critic first
value-loss clipping	a clip band on the critic's change, shipped by some implementations and not by others
early stopping on the approximate KL	the epochs of a batch end once the measured KL passes a threshold

Two studies measured which of these matter. Engstrom et al. (2020) found that without them PPO's ratios leave the clip band freely, and that TRPO given the same details performs about as well as PPO, so much of PPO's advantage comes from the code around its objective. Andrychowicz et al. (2021) trained hundreds of thousands of agents over a grid of such choices and turned the results into recommendations: normalizing observations helped on most tasks, and neither study found value-loss clipping to help. Reading a maintained single-file implementation beside Section 15.2 turns this table into code.

The PPO paper offers a second way to bound the step (Schulman et al. 2017): a KL penalty on the change from the old policy, whose weight is doubled after a batch that overshot a target KL and halved after one that fell short. It did worse than the clip in their experiments, but it is a close relative of the penalty to a fixed reference in Section 15.3.

In distributed training, many actors collect data with copies of the policy that lag the learner by several updates. IMPALA's V-trace target (Espeholt et al. 2018) corrects the n-step targets of Section 15.1 with importance ratios truncated at one. The truncation keeps the variance bounded, and in exchange the targets estimate the value of a policy between the actors' and the learner's.

15.9.2 Value Learning after DQN

The DQN of Section 15.4 needs tens of thousands of CartPole steps, and its maximum over noisy estimates overestimates. Its successors each change one part. A multi-step target puts the n-step return of Section 15.1 inside the Q-learning target, so a reward reaches earlier states in fewer updates. Prioritized replay (Schaul et al. 2016) samples transitions in proportion to their TD error and reweights them to undo the bias this introduces. A dueling network (Wang et al. 2016) splits the output into a state value and per-action advantages, so the shared value learns from the transitions of every action.

Distributional methods learn the whole distribution of the return rather than its mean, with a Bellman equation for distributions (Bellemare et al. 2017). Lyle et al. (2019) showed that in tabular and linear settings the expected values they learn are those of the ordinary update,

so their gains come from how the richer target trains a deep network. Rainbow (Hessel et al. 2018) combined six of these changes, the double estimator of Section 15.4 among them. Its ablation on Atari ranked prioritized replay and multi-step targets as the most important parts, and the double estimator and the dueling head as the least.

Hasselt et al. (2018) measured how often the deadly triad of Section 15.4 causes trouble in deep Q-networks trained under many settings. Unbounded divergence was rare, but estimates that rose for a while above any achievable return were common; multi-step targets and target networks made such episodes rarer. Two recent designs drop the repairs of Section 15.4 altogether. PQN (Gallici et al. 2025) trains on batches from many parallel environments with layer normalization in the network, and uses neither a replay buffer nor a target network. Streaming methods (Elsayed et al. 2024) learn from each transition once, as it arrives, after normalizing observations and rewards and bounding the size of each update. Replay and a frozen copy are therefore one repair of correlated data and moving targets, not the only one.

15.9.3 Continuous Control since 2018

TD3 (Fujimoto et al. 2018) made three changes to DDPG (Lillicrap et al. 2016), whose deterministic actor climbs its critic's errors: the twin critics that Section 15.5 adopted, an actor updated less often than the critics, and noise on the target's action, so that the actor cannot exploit a narrow peak of the critic.

Most progress since has come from taking more gradient steps per environment step, the update ratio of Section 15.5, without the critic's estimates running away. REDQ (X. Chen et al. 2021) keeps a large ensemble of critics and takes the minimum over a random pair of them, which sustains many updates per step. DroQ (Hiraoka et al. 2022) replaces the ensemble by dropout and layer normalization in two critics. CrossQ (Bhatt et al. 2024) is the exception: it matches REDQ at one update per step and has no target networks; its critic uses batch normalization and passes the current and the next state-action pairs through the network in one batch, so that both see the same statistics. BRO (Nauman et al. 2024) scales the critic up to a wide residual network with layer normalization and adds optimistic exploration. The common thread with PQN is normalization inside the critic.

15.9.4 Models, Imagination and Search

The planner of Section 15.6 spends thousands of model evaluations per real step and trusts its model only near the data; the methods below use a learned model differently. Dyna (Sutton 1990) uses it to imagine extra transitions for a model-free learner, the counted model of Section 14.3 driving Q-learning updates between real steps. MBPO (Janner et al. 2019) keeps the imagined rollouts short, a few steps branching from states in the replay buffer, and mixes them into soft actor-critic's batches, so the model's compounding error stays small. REDQ, from the previous topic, matched MBPO's sample efficiency without a model, which suggests that much of MBPO's gain came from more updates per real step.

PETS (Chua et al. 2018) fits an ensemble of probabilistic networks and plans through samples from it with the cross-entropy method. The members agree where the data is dense and disagree elsewhere, so their disagreement marks where a plan should not be trusted. Dreamer (Hafner et al. 2025) learns a recurrent model of the dynamics in a learned latent space, from images, and trains its actor and critic entirely on imagined latent trajectories, with one setting of its hyperparameters across many domains.

Search with a learned value closes the loop of Section 15.6. AlphaZero (Silver et al. 2017, 2018) searches with the rules of the game as its model and a value network at the leaves, and distills the search's choices into the policy. MuZero (Schrittwieser et al. 2020) learns a model that predicts only rewards, values and policies, not observations, and searches in it. TD-MPC2 (Hansen et al. 2024) plans over short horizons in a latent model and closes each plan with a learned value.

A learned model also helps when no new data may be collected, provided the offline learner distrusts it where the data is thin. MOPO (Yu et al. 2020) and MOReL (Kidambi et al. 2020) penalize imagined rewards where an ensemble of models disagrees: the pessimism of Section 15.7, applied to a model.

15.9.5 Offline Methods at Scale

The penalty of Section 15.7 shrinks with the square root of the count of a state-action pair in the log. With continuous actions almost every action a learned policy proposes has a count of zero, so the deep offline methods replace the count by something a network can estimate. They differ in what they constrain:

method	what it constrains	how
BCQ (Fujimoto et al. 2019)	the actions the policy may take	a generative model of the logged actions proposes candidates, and the policy picks among them
CQL (Kumar et al. 2020)	the values	a loss term lowers Q on the policy's actions and raises it on the logged ones
IQL (Kostrikov et al. 2022)	which actions the backup queries	V is fitted to an upper expectile of Q over logged actions only, and the policy is extracted by advantage-weighted regression
TD3+BC (Fujimoto and Gu 2021)	the policy's distance from the data	a cloning term is added to TD3's actor loss

Return-conditioned sequence models, the other family, are discussed in Section 15.7 together with their limits.

Many applications have a log and can also collect a little new data. Cal-QL (Nakamoto et al. 2023) calibrates the pessimistic values so that they never fall below those of a reference policy such as the logging policy, which removes the drop in return that overly low values otherwise cause when online fine-tuning begins. RLPD (Ball et al. 2023) skips offline training and runs an off-policy learner online from the start, with half of every batch drawn from the log and with the normalized critics of the previous topics.

15.9.6 Post-Training Variants

Section 15.8 assembles the post-training update from the group weight of Section 14.7, PPO's clipped ratio and the KL penalty of Section 15.3, and names DAPO, Dr. GRPO and GSPO.

Trained at the scale of reasoning models (Shao et al. 2024; DeepSeek-AI 2025), the update met problems that the one-token task cannot show, and the variants answer them. DAPO (Q. Yu et al. 2025) also averages the loss over all tokens of a batch rather than first within each response, so that a long response counts in proportion to its length, and it shapes the reward of responses cut off at the length limit. Dr. GRPO (Z. Liu et al. 2025) traces two biases to the divisions it removes: dividing by the response length shrinks the per-token penalty of long wrong answers, so training lengthens them, and dividing by the group's standard deviation gives the most weight to prompts whose groups are nearly all right or all wrong. GSPO (C. Zheng et al. 2025) was introduced for the stability of large mixture-of-experts models, whose per-token ratios are especially noisy.

Process rewards change where the reward comes from. Lightman et al. (2024) trained a reward model on human labels of individual reasoning steps, and it judged solutions better than a model trained on final answers alone. A learned process reward gives each step its own credit, as the prefix values of Section 15.8 do, and brings with it the exploitable errors of a learned reward that Section 15.3 measured.

15.9.7 Beyond the Chapter

Three assumptions of these two chapters can fail. When several agents learn at once, each one's environment contains the others, and nothing is stationary; self-play combined with search reached superhuman play in heads-up no-limit poker (Brown and Sandholm 2017). When the agent sees observations rather than the state, the Markov property of Section 14.1 fails, and the policy needs memory, a recurrent network or a belief over states (Kaelbling et al. 1998). When tasks change, an agent can learn to adapt to a new task from a few episodes (meta-learning), or learn sub-policies that act over many steps together with a policy that chooses among them (hierarchical learning). Murphy's overview (Murphy 2025) surveys all three.

15.9.8 A Project

Section 15.8 assembles the post-training update and applies it to a single group. This project builds the training loop around it and checks it against what the chapter can compute exactly.

1. **The task.** Use the four prompts of the verifier task, answered with two digit tokens as in Section 15.8, so that `04`, `12`, `64` and `81` are the correct responses. The policy is a table of preferences for the tens digit per prompt and for the units digit per prompt and tens digit; the reference is uniform.
2. **The update.** Extend `grpo_step` to two-token responses. For each prompt, sample a group of 16 responses, store their token log-probabilities, and standardize the rewards within the group. Then run E epochs of gradient ascent with step size 1 on the clipped per-token surrogate with $\epsilon_{\text{clip}} = 0.2$, averaged over the tokens of the group, minus β times the k_3 term of Section 15.8; record the clip fraction of each epoch.
3. **The checks.** In the first epoch every ratio equals 1 and the clip fraction is zero, so with $E = 1$ the update is the REINFORCE step with group-standardized weights.
4. **The bar for reuse.** With $\beta = 0$ and the same step size, four epochs per group should reach a success rate of 0.9 with about half the sampled responses that one epoch needs, on every seed you try, while the clip fraction rises from the first epoch to the fourth.
5. **The bar for the penalty.** On the one-token verifier task with the lenient checker of Section 15.8, use the leave-one-out weight of Section 14.7 and $\beta = 1/2$, and train until

the policy settles. Under the k_3 loss the hedge should keep about half of the probability on `8*8`, the maximizer of the objective with the KL in the other direction (exercise 5); with the penalty folded into the reward it should stay under two percent, the tilted optimum. Explain why the leave-one-out weight is needed for these two numbers, and what group standardization would change.

Chapter 16

Generative Adversarial Networks

Almost every model in this book was trained by likelihood: a classifier raises the probability of the correct label, a language model that of the next token. The reward model of Section 15.3 learned differently, from which of two answers a judge preferred, through the Bradley–Terry model, which turns the difference of two scores into the probability that one answer wins. Now consider a network that turns a hundred random numbers into an image in one forward pass. Nothing in it computes the probability of the image it returns, so there is no likelihood to raise. Can a learned comparison take its place?

Goodfellow et al. (2014) proposed that it can. A second network, the critic, learns to tell generated samples from real ones, and the generator learns to make that hard. The critic's output tells the generator where its samples are too dense and where too sparse, although neither density is ever written down.

Diffusion and flow models (Chapter 17) have since replaced stand-alone GANs for most image synthesis, yet the learned comparison survives: one-step generators distilled from diffusion models, the autoencoders on which many image generators run, and many speech vocoders all train with an adversarial term.

Seven questions organize the chapter. What does the critic compute, and what does the generator then minimize (Section 16.1)? Which losses and critics keep a gradient when the generated samples miss the data (Section 16.2)? What changes when the critic ranks a real sample against a generated one (Section 16.3)? Why does gradient training circle the equilibrium, and what makes it converge (Section 16.4)? Which network turns noise into an image, and how do we score the images (Section 16.5)? How do we ask for a bird, and check that we got one (Section 16.6)? Why does squared error blur an ambiguous prediction where a learned comparison does not (Section 16.7)? The experiments, in PyTorch and JAX, run on Fashion-MNIST garments, planar Gaussians whose densities we know, game sprites and CIFAR-10.

Resources and Further Reading

These resources accompany particular sections; Section 16.8 turns to the methods the chapter does not teach.

- The NIPS 2016 tutorial (Goodfellow 2016) gives the original intuition and the field in 2016, a companion to Section 16.1.
- GAN Lab (Kahng et al. 2018) runs two-dimensional games in a browser and shows the critic's decision surface as training proceeds.
- The GAN stability repository of Mescheder et al. (2018) implements the Dirac-GAN and the R_1/R_2 penalties of Section 16.4.
- The R3GAN implementation (Huang et al. 2024) trains the loss and backbone of Section 16.4 and Section 16.5 at full scale.
- StyleGAN3 (Karras et al. 2021) is a mature StyleGAN implementation with full-scale FID and KID tools.
- Lucic et al. (2018) find that, given equal tuning budgets, many objectives of Section 16.2 attain similar scores.
- ganhacks, a 2016 training checklist, shows which stabilizers the penalties of Section 16.4 later replaced.

16.1 Generative Adversarial Networks

Suppose we want a network that draws a Fashion-MNIST garment from a hundred random numbers in a single forward pass. It produces images but no probabilities, so the likelihood training of the earlier chapters does not apply. How can such a network be trained?

We have met the way out before. To correct for covariate shift in Section 3.7.2, we trained a classifier to tell training inputs from test inputs and read the ratio of their two densities off its logit, although neither density was ever written down. A *generative adversarial network* (GAN) (Goodfellow et al. 2014) trains the same kind of classifier on real and generated samples, and then changes the generator until the classifier fails, as a counterfeiter improves until the police can no longer tell forged notes from real ones.

This section asks what that game measures and whether it gives the generator a usable signal. We solve the classifier's problem exactly and find the quantity the generator then minimizes. We then train a generator of Fashion-MNIST garments, and check the predictions of the analysis on a two-dimensional Gaussian whose densities we know.

16.1.1 From Likelihood to Comparison

16.1.1.1 Implicit Generators

Maximum likelihood needs $\log p_\theta(x)$, the model's log-density at each data point (Section 36.6.2.1). The generators of this chapter are *implicit*: they draw a latent variable $z \sim \mathcal{N}(0, I_{d_z})$ and output $x' = G(z)$, where G is a neural network. When the latent dimension d_z is smaller than the data dimension, the samples lie on a lower-dimensional surface and have no density at all; when the dimensions match, evaluating the density would require an invertible G with a tractable Jacobian.

Nor is likelihood what we want from a sampler. Consider a model that mixes the data density with noise,

$$\tilde{p} \;=\; 0.01\, p_{\text{data}} + 0.99\, p_{\text{noise}}. \tag{16.1}$$

Since $\tilde{p}(x) \geq 0.01\, p_{\text{data}}(x)$ everywhere, the expected log-likelihood of $\tilde{p}$ is within $\log 100 \approx 4.6$ nats of the best achievable by any model, a small shortfall for images, whose log-

likelihoods run to thousands of nats. Yet 99 percent of its samples are noise (Theis et al. 2016). Judging a sampler requires looking at its samples.

16.1.1.2 A Classifier between Real and Generated Samples

A *two-sample test* asks whether two sets of examples came from the same distribution, and one way to run it is to train a classifier to tell them apart. A GAN turns this test into a training signal. Draw a label $y \sim \text{Bernoulli}(\frac{1}{2})$. If $y = 1$, draw x from the data density p; if $y = 0$, draw x from the generator density q. The marginal density of x is the balanced mixture $m = (p + q)/2$.

The classifier is a function $D : \mathcal{X} \to \mathbb{R}$ that outputs a *realness logit*: with $\sigma(t) = 1/(1 + e^{-t})$, the value $\sigma(D(x))$ estimates $P(y = 1 \mid x)$. Goodfellow et al. (2014) called D the *discriminator*. We call it the *critic*, the name used for its successors later in the chapter, which score samples without classifying them.

Training alternates between the two networks, as Figure 16.1 shows. The critic is fitted to separate a batch of data from a batch of generated samples. The generator then changes so that its samples become harder to separate, receiving its gradient through the critic.

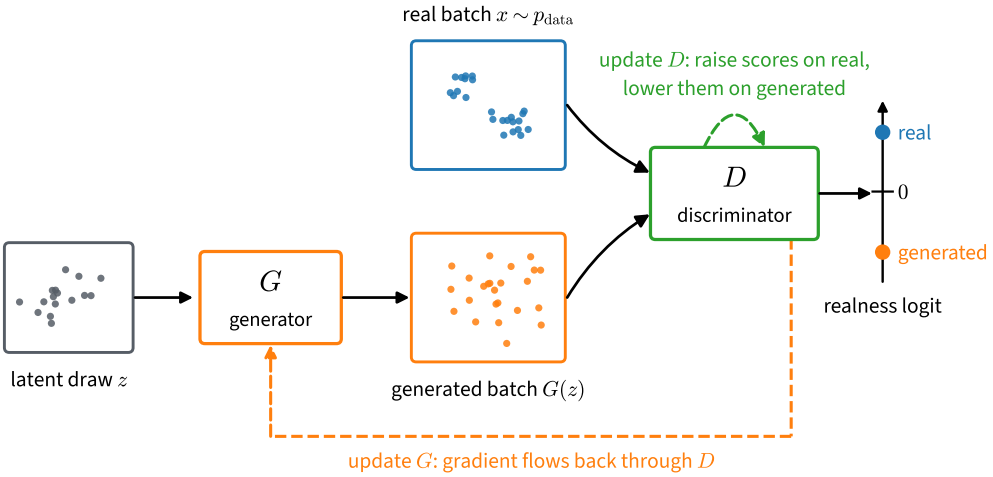


Figure 16.1: Adversarial training. The generator G turns latent draws z into generated samples, and the critic D scores them and a batch of real data. The critic raises its scores on data and lowers them on generated samples; the generator, through the critic's gradients, makes the two batches harder to tell apart.

16.1.1.3 Notation

Table 16.1 collects the chapter's symbols, each with one meaning throughout; letters defined inside an exercise are local to it.

Table 16.1: Notation of this chapter.

symbol	meaning	first used	papers write
p, q, $m = \frac{1}{2}(p+q)$	data, generator and mixture densities	Section 16.1	
$x \sim p$, $x' \sim q$; y	a real and a generated sample; the source label, 1 for real	Section 16.1	y or $\hat{x}$ for x'
z, G, d_z; d_x	latent variable, generator, latent dimension; data dimension	Section 16.1	k for d_z, d for d_x
D, V, V_ℓ	the critic's realness logit; the value of the log-loss game, of the game with payoff ℓ	Section 16.1	discriminator; $-D$ in Huang et al. (2024)
$\rho = p/q$, $\lambda = \log \rho$	density ratio and its logarithm	Section 16.1	
μ; b; δ_θ; $d(p, q)$	a mean; an additive shift; a point mass at θ; a discrepancy	Section 16.1	δ for a separation of means
w_{sat}, w_{ns}, w_{Rp}	per-sample weights of the generator update	Section 16.1	not named
$\eta = p/(p+q)$	class posterior	Section 16.2	η for a step size, h here
ℓ, L, Δ_ℓ	payoff of a score, its conditional Bayes risk, its Bayes-risk gap	Section 16.2	the loss $-\ell$
$T \in \mathcal{T}$, ℓ_p, ℓ_q; $\mathcal{F}$	a critic and its class, scores of real and generated samples; the class of an integral probability metric	Section 16.2	f for its test functions
f, D_f; k, s	divergence function, f-divergence; kernel, bandwidth	Section 16.2	σ or ℓ for s
Φ, d_{Rp}	pairing objective, its value at the optimal critic	Section 16.3	$2\, d_{\text{Rp}}$
(a_1, a_2), P, Q, π	a real–generated pair, its orderings $p \otimes q$ and $q \otimes p$, a pairing permutation	Section 16.3	

symbol	meaning	first used	papers write
$\mathcal{E}_{\text{GAN}}$, $\mathcal{E}_{\text{diag}}$, $\mathcal{E}_{\text{all}}$	classical, paired and all-pairs losses at the best response	Section 16.3	not named
v, ϵ	direction and size of a change $q = p\,(1 + \epsilon v)$	Section 16.3	h for v
γ, R_1, R_2	penalty weight, zero-centered gradient penalties	Section 16.4	
h, $\nu_{1,2}$, ψ, $\mathbf{u}$; ω	Dirac-GAN step size, eigenvalues, critic slope, field; a wavenumber	Section 16.4	$\lambda_{1,2}$, v; k
ϕ	a feature map	Section 16.5	
c, e_c, $a(c)$, ψ_0, ζ_c	condition, its embedding, modulation scale, unconditional head, per-class constant	Section 16.6	$\gamma(c)$, $\beta(c)$ for scale and shift
g; ξ; ϑ	predictor of x from c; standard normal noise; a phase	Section 16.7	f; ϵ; φ

Divergences and entropies follow the appendix and are in nats: D_{KL} is the Kullback–Leibler divergence, $D_{\text{J}} = D_{\text{KL}}(p \,\|\, q) + D_{\text{KL}}(q \,\|\, p)$ its symmetrized form, JS the Jensen–Shannon divergence, H an entropy and $H_b(t)$ that of a coin with bias t.

16.1.2 The Log-Loss Game

The natural loss for a probabilistic classifier is the log loss. The critic's log-likelihood on the balanced labeling problem, without a factor $\frac{1}{2}$, is

$$V(D) \;=\; \mathbb{E}_{x\sim p}\big[\log \sigma(D(x))\big] + \mathbb{E}_{x'\sim q}\big[\log \sigma(-D(x'))\big], \tag{16.2}$$

the value function of Goodfellow et al. (2014). The critic maximizes V; the generator influences only the second term and minimizes it, so training solves the *minimax* problem $\min_G \max_D V$. Because maximizing V fits the best classifier, $\max_D V$ measures how well p and q can be told apart.

16.1.2.1 The Optimal Critic

The maximization over D separates into independent scalar problems. The integrand of Equation 16.2 at a point x is $p(x)\log\sigma(D(x)) + q(x)\log\sigma(-D(x))$, which depends on D only through the number $D(x)$. Writing $u = \sigma(D(x))$, the problem at each point is to maximize $p\log u + q\log(1-u)$, a strictly concave function of u whose stationarity condition $p/u = q/(1-u)$ has the unique solution

$$\sigma(D^\star(x)) = \frac{p(x)}{p(x)+q(x)} = P(y=1 \mid x), \qquad D^\star(x) = \log\frac{p(x)}{q(x)} = \lambda(x). \tag{16.3}$$

Thus the optimal critic returns the Bayes posterior of the labeling problem, and its logit is the logarithm $\lambda = \log\rho$ of the *density ratio* $\rho = p/q$. Although q has no computable density, a classifier estimates the one function of q the objective needs, just as the domain classifier of covariate shift did. The objective also fixes the critic's additive constant: shifting $D^\star$ by any $b \neq 0$ strictly decreases V (Exercise 2b), so $D(x) > 0$ says that x is more likely real than generated.

For two unit Gaussians a distance μ apart, $p = \mathcal{N}(0,1)$ and $q = \mathcal{N}(\mu,1)$, the optimal critic can be written down by hand:

$$\lambda(x) = -\frac{x^2}{2} + \frac{(x-\mu)^2}{2} = \mu\Big(\frac{\mu}{2} - x\Big),$$

a linear function of x, positive left of the midpoint $\mu/2$. A logistic regression $D(x) = wx+b$ therefore contains the optimal critic. When the two Gaussians also differ in their covariances, λ is quadratic and a nonlinear critic is needed.

16.1.2.2 The Value of the Game

Substituting the best response into the objective identifies what the generator minimizes. With $\sigma(D^\star) = p/(2m)$ and $\sigma(-D^\star) = q/(2m)$,

$$V(D^\star) = \int p\log\frac{p}{2m} + \int q\log\frac{q}{2m} = D_{\mathrm{KL}}(p\,\|\,m) + D_{\mathrm{KL}}(q\,\|\,m) - 2\log 2.$$

The two divergences define the Jensen–Shannon divergence $\mathrm{JS}(p,q) = \frac{1}{2}D_{\mathrm{KL}}(p\,\|\,m) + \frac{1}{2}D_{\mathrm{KL}}(q\,\|\,m)$. Hence

$$\max_D V(D) = 2\,\mathrm{JS}(p,q) - 2\log 2. \tag{16.4}$$

Against an optimal critic, the generator minimizes the Jensen–Shannon divergence, which vanishes exactly when $q = p$. Equivalently, the critic's smallest per-sample negative log-likelihood is $\log 2 - \mathrm{JS}(p,q)$: chance level minus the divergence. For the two unit Gaussians, JS is 0.11 nats at $\mu = 1$, 0.34 at $\mu = 2$ and 0.63 at $\mu = 4$; at $\mu = 8$ it equals $\log 2 \approx 0.6931$ to four digits.

Writing $D_{\mathrm{KL}}(p\,\|\,m) = -H(p) - \int p\log m$, and likewise for q, and collecting the mixture terms into $-H(m)$ shows what the game measures:

$$\mathrm{JS}(p,q) \;=\; H(m) - \frac{H(p)+H(q)}{2} \;=\; I(x;y). \tag{16.5}$$

The first term is the uncertainty about a sample whose source is unknown, the second the average uncertainty once the source is known. Their difference is the mutual information

between a sample $x \sim m$ and its source label y: the number of nats that one sample provides about its source. A binary label carries at most $\log 2$ nats, so $0 \le \mathrm{JS}(p, q) \le \log 2$ (Section 37.3 proves both bounds), and the upper end is reached exactly when the supports of p and q are disjoint and every sample identifies its source.

16.1.2.3 Separated Supports

The ceiling has a cost: once the supports separate, the value stops depending on how far apart they are. Take two point masses on the line, $p = \delta_0$ and $q_\theta = \delta_\theta$. For every $\theta \neq 0$ their supports are disjoint, $\mathrm{JS}(p, q_\theta) = \log 2$, and the value does not depend on θ. A constant objective supplies no gradient, even when the generator lies arbitrarily close to the data.

The case is not exotic. A generator that maps a d_z-dimensional latent variable into a higher-dimensional space places its samples near a d_z-dimensional surface, and natural images also concentrate near a low-dimensional set, so the two supports are typically disjoint or nearly so (Arjovsky and Bottou 2017). An optimal critic then classifies perfectly, and the generator learns nothing from the value of the game.

16.1.3 The Generator's Gradient

The value describes the game at the critic's best response, but training moves the generator by gradient steps with the critic held fixed. We now follow that gradient.

16.1.3.1 Two Weights

The generator influences only the second term of Equation 16.2, so playing the minimax game literally means minimizing $\mathbb{E}_z[\log \sigma(-D(G(z)))]$ over the generator's parameters θ, with the critic fixed. Differentiating through $x' = G(z)$ with the identity $\frac{d}{du} \log \sigma(-u) = -\sigma(u)$ gives

$$\nabla_\theta \, \mathbb{E}_z\big[\log \sigma(-D(G(z)))\big] = -\,\mathbb{E}_z\big[\sigma(D(x'))\, \nabla_\theta D(G(z))\big].$$

A descent step therefore moves each generated sample in the direction that raises its critic score, with weight $\sigma(D(x'))$, the critic's probability that the sample is real. For a sample the critic rejects, $D(x') \ll 0$ and this weight is exponentially small. The loss has *saturated*: the samples the critic rejects most confidently contribute least to the update.

Goodfellow et al. (2014) therefore use a different generator loss in practice: rather than minimizing the probability of being called fake, the generator maximizes that of being called real, minimizing $-\mathbb{E}_z[\log \sigma(D(G(z)))]$. With $\frac{d}{du} \log \sigma(u) = \sigma(-u)$,

$$\nabla_\theta \Big(-\mathbb{E}_z\big[\log \sigma(D(G(z)))\big]\Big) = -\,\mathbb{E}_z\big[\sigma(-D(x'))\, \nabla_\theta D(G(z))\big].$$

The two losses move each sample in the *same* direction $\nabla_\theta D$ and differ only in the weight attached to it:

$$w_{\text{sat}}(x') = \sigma(D(x')), \qquad w_{\text{ns}}(x') = \sigma(-D(x')) = 1 - \sigma(D(x')). \tag{16.6}$$

The *non-saturating* weight w_{ns} is near one where the saturating weight is near zero, as Figure 16.2 shows. Their ratio is $w_{\text{ns}}/w_{\text{sat}} = e^{-D(x')}$: a sample with score -3 receives 20 times more weight under the non-saturating loss, one with score -5 about 148 times more.

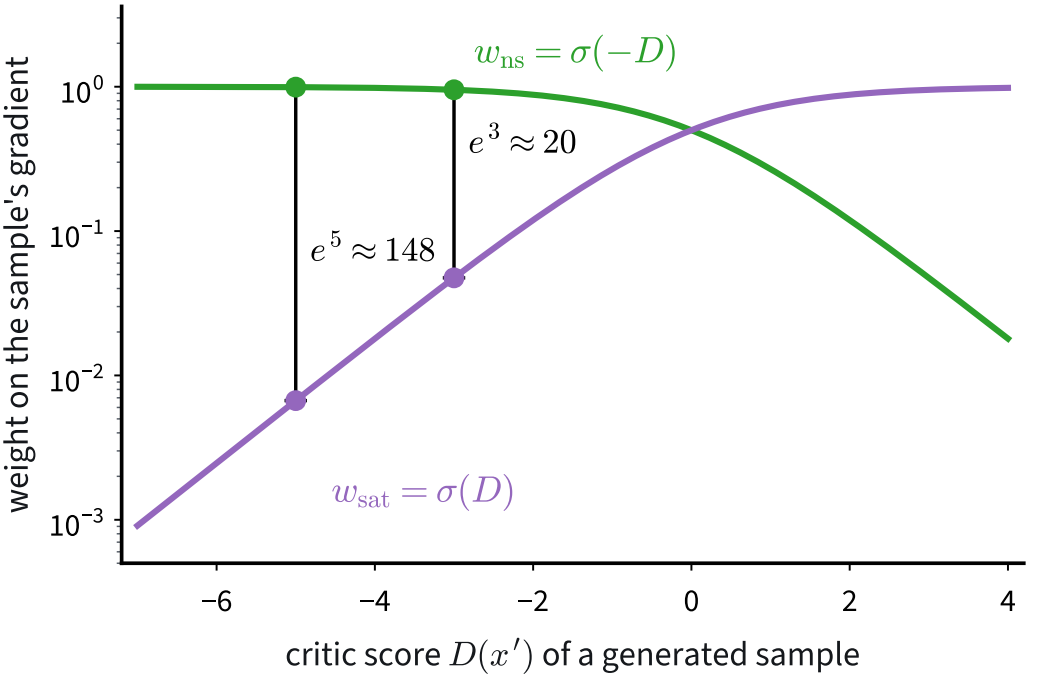


Figure 16.2: The two generator weights against the critic score $D(x')$ of a generated sample, on a logarithmic axis. For a rejected sample the saturating weight $\sigma(D)$ falls exponentially, while the non-saturating weight $\sigma(-D)$ stays near one; their ratio is e^{-D}.

16.1.3.2 The Same Fixed Point

Changing the weight does not change the population minimizer against an optimal critic. Substituting $D^\star = \lambda$ into the non-saturating loss and using $\sigma(\lambda) = p/(2m)$,

$$-\mathbb{E}_{x'\sim q}\big[\log \sigma(\lambda(x'))\big] = \log 2 + D_{\text{KL}}(q \,\|\, p) - D_{\text{KL}}(q \,\|\, m) \;\geq\; \log 2 + \tfrac{1}{2}\, D_{\text{KL}}(q \,\|\, p). \tag{16.7}$$

The inequality is the convexity of $D_{\text{KL}}(q \,\|\, \cdot)$ at the midpoint m: $D_{\text{KL}}(q \,\|\, m) \leq \frac{1}{2} D_{\text{KL}}(q \,\|\, p) + \frac{1}{2} D_{\text{KL}}(q \,\|\, q)$, and the last term is zero. The bound exceeds $\log 2$ unless $q = p$, so with an optimal critic the non-saturating loss is minimized only at $q = p$.

A gradient step, however, holds the critic fixed rather than re-optimizing it. With the critic fixed at $D = \lambda$, the non-saturating update descends $D_{\text{KL}}(q \,\|\, p) - 2\,\text{JS}(p, q)$ (Arjovsky and Bottou 2017), an f-divergence whose only minimizer is again $q = p$ (Exercise 9).

16.1.3.3 Mode Dropping

That divergence inherits the asymmetry of its first term, the reverse Kullback–Leibler divergence $D_{\mathrm{KL}}(q \,\|\, p)$, which charges the generator heavily for placing mass where the data have little, but lightly for missing mass where the data have much. Take p an equal mixture of two well-separated modes and q a copy of one of them. On the covered mode $p = q/2$, so $D_{\mathrm{KL}}(q \,\|\, p) = \log 2 \approx 0.69$ nats and $\mathrm{JS}(p, q) \approx 0.22$ nats, while the forward divergence $D_{\mathrm{KL}}(p \,\|\, q)$ is infinite. A generator that covers only some of the data's modes is said to suffer *mode dropping*. Gradient steps rarely leave such a state: at the best response the score on the covered mode is the constant $\log(p/q) = -\log 2$, so $\nabla_\theta D$ vanishes there and neither weight pushes toward the missing mode.

A critic away from its best response shows a second blindness. Both weights read a sample's own score $D(x')$ against the zero level that the log-loss game fixes, and nothing in them compares the sample with real data. A generator whose samples all score above zero receives little push, whatever the data's other modes do. Later in the chapter, a critic that ranks a generated sample against a real one replaces this threshold with a comparison.

16.1.4 The Game on Fashion-MNIST

In practice both players are networks, updated in turn by gradient steps on minibatches. We write these updates once and use them to train a generator of images.

16.1.4.1 The Update Rules

One training iteration is two half-steps, one per player. The critic's update ascends Equation 16.2 on a minibatch as logistic regression, with label one for data and zero for generated samples, while the generator is held fixed. Both update functions are saved to the `d2l` library for the image networks of later sections.

```
def update_D(X, Z, net_D, net_G, trainer_D):
    """Update the critic; return the summed log loss, halved."""
    trainer_D.zero_grad()
    real_Y = net_D(X)
    # The generator is the fixed player here: detach its output
    fake_Y = net_D(net_G(Z).detach())
    loss_D = (F.binary_cross_entropy_with_logits(
                  real_Y, torch.ones_like(real_Y), reduction='sum') +
              F.binary_cross_entropy_with_logits(
                  fake_Y, torch.zeros_like(fake_Y), reduction='sum')) / 2
    loss_D.backward()
    trainer_D.step()
    return loss_D
```

The generator's update implements the non-saturating loss of Equation 16.6: generated samples are fed to the critic under the label *one*, so that the cross-entropy is $-\mathbb{E}[\log \sigma(D(G(z)))]$.

```
def update_G(Z, net_D, net_G, trainer_G):
    """Update the generator on the non-saturating loss; return its sum."""
    trainer_G.zero_grad()
```

```
    # net_D changed in its half-step, so recompute the scores
    fake_Y = net_D(net_G(Z))
    loss_G = F.binary_cross_entropy_with_logits(
        fake_Y, torch.ones_like(fake_Y), reduction='sum')
    loss_G.backward()
    trainer_G.step()
    return loss_G
```

The loop `train_gan` alternates the two half-steps over minibatches, drawing a fresh latent batch of dimension `latent_dim` for each, and returns each player's per-sample loss for every epoch; an optional callback `on_epoch` receives the number of each finished epoch. Both players use Adam with $\beta_1 = 0.5$ instead of the default 0.9, the DCGAN setting of Radford et al. (2015): a shorter momentum lets each player respond sooner to the other's latest move.

```
def train_gan(net_D, net_G, data_iter, num_epochs, latent_dim, lr_D, lr_G,
              step_G=update_G, seed=None, on_epoch=None):
    """Alternate update_D and step_G over the minibatches with Adam
    (beta_1 = 0.5), calling on_epoch(epoch) after each epoch if given;
    return the per-sample (loss_D, loss_G) of each epoch."""
    if seed is not None:
        torch.manual_seed(seed)
    trainer_D = torch.optim.Adam(net_D.parameters(), lr=lr_D,
                                 betas=(0.5, 0.999))
    trainer_G = torch.optim.Adam(net_G.parameters(), lr=lr_G,
                                 betas=(0.5, 0.999))
    history = []
    for epoch in range(1, num_epochs + 1):
        loss_D, loss_G, n = 0, 0, 0  # summed on the device, read once
        for X, *_ in data_iter:  # labels, if any, are ignored
            Z = torch.randn(X.shape[0], latent_dim, device=X.device)
            loss_D += update_D(X, Z, net_D, net_G, trainer_D).detach()
            loss_G += step_G(Z, net_D, net_G, trainer_G).detach()
            n += X.shape[0]
        history.append(torch.stack([loss_D, loss_G]) / n)
        if on_epoch is not None:
            on_epoch(epoch)
    return torch.stack(history).cpu().numpy()
```

16.1.4.2 Garments from Noise

Fashion-MNIST, the 28×28 grayscale images of ten kinds of clothing from Section 3.2, gives a first test on images. Both players are multilayer perceptrons with leaky ReLU units, which pass negative inputs with slope 0.2 instead of zeroing them. The generator maps 100 latent numbers through hidden layers of 256 and 512 units to 784 pixels, with a final tanh for the range $[-1, 1]$ to which we scale the images. The critic mirrors it, from 784 inputs through 512 and 256 units to one logit.

The function `make_mlp` builds such a perceptron from its list of layer widths, with the activation `act` between consecutive linear layers. Its weights start uniform in $\pm 1/\sqrt{n_{\text{in}}}$ for a layer with n_{in} inputs, and its offsets at zero. It is saved for the small networks of later sections.

```
def make_mlp(sizes, act):
    """Linear layers of the given widths with the activation module act
    between them; weights uniform in +-1/sqrt(fan_in), zero biases."""
    layers = []
    for d_in, d_out in zip(sizes[:-1], sizes[1:]):
        linear = nn.Linear(d_in, d_out)  # weights uniform by default
        nn.init.zeros_(linear.bias)
        layers += [linear, act]
    return nn.Sequential(*layers[:-1])
```

```
torch.manual_seed(0)
fmnist = d2l.FashionMNIST()
# All 60,000 training images as rows of 784 pixels in [-1, 1]
images = fmnist.train.data.reshape(-1, 784).float() / 127.5 - 1
image_iter = d2l.load_array((images,), batch_size=64)
leaky = nn.LeakyReLU(0.2)
net_G_img = nn.Sequential(make_mlp([100, 256, 512, 784], leaky), nn.Tanh())
net_D_img = make_mlp([784, 512, 256, 1], leaky)
```

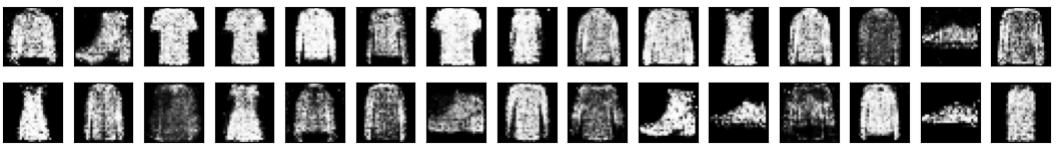
Both players learn at the DCGAN's rate of $2 \cdot 10^{-4}$, on minibatches of 64, for twenty epochs (almost 19,000 iterations).

```
history_img = train_gan(net_D_img, net_G_img, image_iter, num_epochs=20,
                        latent_dim=100, lr_D=2e-4, lr_G=2e-4, seed=1)
print(f'last epoch: critic loss {history_img[-1, 0]:.2f}, '
      f'generator loss {history_img[-1, 1]:.2f}')
```

```
last epoch: critic loss 0.52, generator loss 1.37
```

Thirty samples from fresh latent draws:

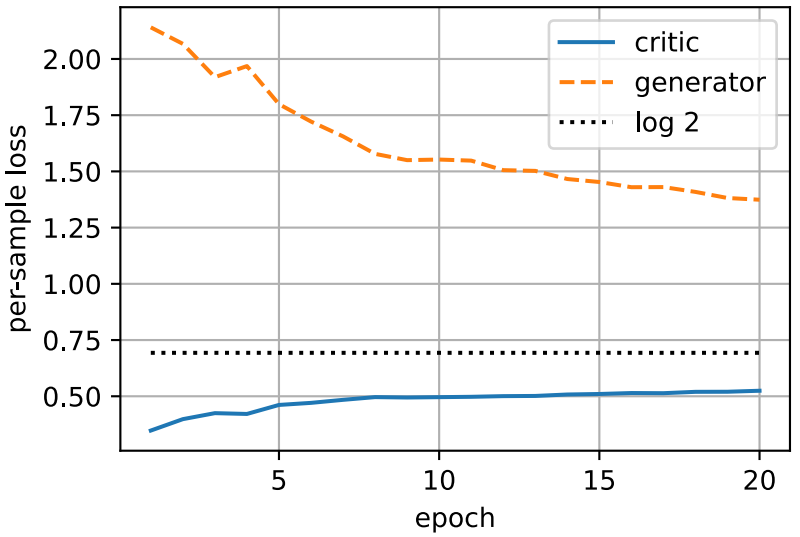
```
with torch.no_grad():
    fake_img = net_G_img(torch.randn(30, 100))
d2l.show_images(fake_img.reshape(30, 28, 28), 2, 15, scale=0.9, cmap='gray');
```



The samples are recognizable garments: shirts and other tops, dresses and shoes. Most outlines are clear, but the interiors are grainy and speckled rather than blurred; neither network knows that neighboring pixels belong together, which the convolutional networks of Section 16.5 add. The two losses show how far the game is from its end.

```
epochs = list(range(1, len(history_img) + 1))
d2l.plot(epochs, [history_img[:, 0], history_img[:, 1],
                  [math.log(2)] * len(epochs)],
```

```
        'epoch', 'per-sample loss', legend=['critic', 'generator', 'log 2'],
        fmts=('-', '--', 'k:'), figsize=(4.5, 3))
```



Within the first epoch the critic pulls ahead: its per-sample loss falls well below $\log 2$ and the generator's rises well above it. Both then drift slowly toward $\log 2$, but after twenty epochs the critic's loss is still about 0.5 and the generator's about 1.4. The critic still tells real from generated garments with some confidence, so at this budget the game has not reached its equilibrium, where both losses would sit at $\log 2$.

16.1.5 The Game on a Gaussian

The garments cannot be scored in nats: neither density can be evaluated, so we can say neither how far q is from p nor whether the critic computes λ. We therefore return to a case where the densities are known. With Gaussian data and a linear generator, p, q and λ have closed forms, and three predictions can be checked exactly: the critic's loss approaches $\log 2$ near the fixed point, a trained critic approximates λ, and a confidently rejected generator keeps learning with the non-saturating loss but stalls with the saturating one.

16.1.5.1 A Linear Generator with Known Densities

The data are a standard Gaussian pushed through a fixed linear map, $x = zA + b$ with $z \sim \mathcal{N}(0, I_2)$, so $p = \mathcal{N}(b, A^\top A)$: a thin tilted ellipse, whose orientation and two very different scales the generator must learn. Training uses a fixed sample of one thousand draws.

```
torch.manual_seed(0)
Z = torch.normal(0.0, 1.0, (1000, 2))
A = torch.tensor([[1.0, 2.0], [-0.1, 0.5]])
b = torch.tensor([1.0, 2.0])
data = Z @ A + b
```

```
print(f'covariance of the data distribution:\n{A.T @ A}')
```

```
covariance of the data distribution:
tensor([[1.0100, 1.9500],
        [1.9500, 4.2500]])
```

The generator is a single linear layer, so its output is Gaussian with a mean and covariance read off the weights. The critic has two hidden layers of 16 leaky ReLU units, which keep its score surface sloped, so the generator almost always receives a direction (Exercise 5 replaces them by tanh units). A factory function builds both networks from a fixed seed, so that later cells can restart from the same initialization.

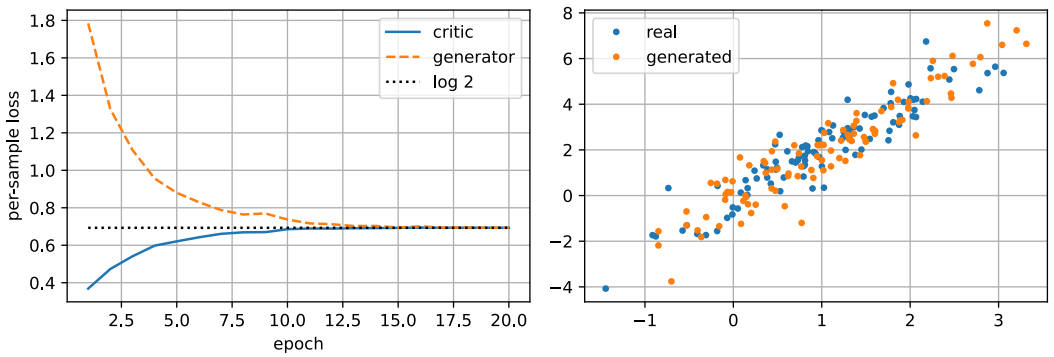
```
def make_nets(seed=7):
    """The linear generator and the critic, all weights and offsets drawn
    from N(0, 0.02^2) by a generator seeded with seed."""
    gen = torch.Generator().manual_seed(seed)
    net_G = nn.Sequential(nn.Linear(2, 2))
    net_D = nn.Sequential(nn.Linear(2, 16), nn.LeakyReLU(0.2),
                          nn.Linear(16, 16), nn.LeakyReLU(0.2),
                          nn.Linear(16, 1))
    for w in [*net_G.parameters(), *net_D.parameters()]:
        nn.init.normal_(w, 0, 0.02, generator=gen)
    return net_G, net_D
```

We train for twenty epochs in minibatches of eight, giving the critic ten times the generator's learning rate so that its scores stay close to their best response.

```
data_iter = d2l.load_array((data,), batch_size=8)
net_G, net_D = make_nets()
history = train_gan(net_D, net_G, data_iter, num_epochs=20, latent_dim=2,
                    lr_D=0.05, lr_G=0.005, seed=1)
with torch.no_grad():
    fake = net_G(torch.normal(0, 1, (200, 2)))
```

The left panel tracks both losses against the dotted line at $\log 2$; the right panel overlays generated samples on the data.

```
fig, axes = d2l.plt.subplots(1, 2, figsize=(9, 3.2))
epochs = list(range(1, len(history) + 1))
d2l.plot(epochs, [history[:, 0], history[:, 1], [math.log(2)] * len(epochs)],
         'epoch', 'per-sample loss', legend=['critic', 'generator', 'log 2'],
         fmts=('-', '--', 'k:'), axes=axes[0])
d2l.plot([data[:100, 0], fake[:100, 0]], [data[:100, 1], fake[:100, 1]],
         legend=['real', 'generated'], fmts=('C0.', 'C1.'), axes=axes[1])
fig.tight_layout()
```



Unlike on the garments, the critic's loss rises to $\log 2$ within about ten epochs and then stays near it, apart from an occasional epoch in which both losses jump above it and fall back. That is what it should do if JS approaches zero, since the best critic's per-sample loss is $\log 2 - \mathrm{JS}(p, q)$. The generator's loss approaches the same level from above, the minimum of the non-saturating loss by Equation 16.7, and the generated samples fill the data's ellipse. Still, a critic loss of $\log 2$ shows only that this critic can no longer separate the samples, not that $q = p$, so we check the fit independently.

16.1.5.2 The Fit in Nats

A linear generator $G(z) = zA_q + b_q$ produces the Gaussian $q = \mathcal{N}(b_q, A_q^\top A_q)$, and between Gaussians in d_x dimensions the Kullback–Leibler divergence has the closed form (Kullback 1959)

$$D_{\mathrm{KL}}(\mathcal{N}(\mu_0, \Sigma_0) \,\|\, \mathcal{N}(\mu_1, \Sigma_1)) = \tfrac{1}{2}\Big[\operatorname{tr}(\Sigma_1^{-1}\Sigma_0) + (\mu_1 - \mu_0)^\top \Sigma_1^{-1}(\mu_1 - \mu_0) - d_x + \log \tfrac{\det \Sigma_1}{\det \Sigma_0}\Big], \tag{16.8}$$

the multivariate form of the univariate formula of Section 37.1. We report $D_{\mathrm{KL}}(q \,\|\, p)$ at initialization and after training.

```
def generator_gaussian(net_G):
    A_qT, b_q = net_G[0].weight.detach(), net_G[0].bias.detach()
    return b_q, A_qT @ A_qT.T  # nn.Linear stores A_q.T

def kl_gaussians(mu0, S0, mu1, S1):
    diff = mu1 - mu0
    return 0.5 * (torch.trace(torch.linalg.solve(S1, S0))
                  + diff @ torch.linalg.solve(S1, diff) - len(mu0)
                  + torch.logdet(S1) - torch.logdet(S0))

mu_p, Sigma_p = b, A.T @ A
for name, net in [('at initialization', make_nets()[0]),
                  ('after training   ', net_G)]:
    mu_q, Sigma_q = generator_gaussian(net)
    print(f'KL(q || p) {name}: '
          f'{kl_gaussians(mu_q, Sigma_q, mu_p, Sigma_p):.3f} nats')
```

```
KL(q || p) at initialization: 7.501 nats
KL(q || p) after training   : 0.010 nats
```

The divergence falls from about eight nats at initialization to well below one, typically a few hundredths of a nat or less. The training loop never evaluates the generator's density, yet the critic supplies enough information to fit its parameters.

16.1.5.3 The Learned Density Ratio

Equation 16.3 says that the best response to a *fixed* generator computes $\lambda = \log(p/q)$. After the run above, λ is close to zero everywhere and even a constant critic would look accurate, so we set a generator by hand, $A_q = [[1, 2], [-0.5, 0.5]]$ and $b_q = (1.5, 1.5)$, a Gaussian q at $D_{\mathrm{KL}}(q \,\|\, p) \approx 3.4$ nats from the data, and train a fresh critic against it alone for 2,000 steps.

```
net_G_q, net_D_q = make_nets()  # a fresh critic
with torch.no_grad():  # the hand-set generator; nn.Linear stores A_q.T
    net_G_q[0].weight.copy_(torch.tensor([[1.0, -0.5], [2.0, 0.5]]))
    net_G_q[0].bias.copy_(torch.tensor([1.5, 1.5]))
trainer_D = torch.optim.Adam(net_D_q.parameters(), lr=0.01)
torch.manual_seed(2)
for _ in range(2000):  # critic-only steps against the frozen generator
    X = data[torch.randint(0, len(data), (256,))]
    Zb = torch.normal(0, 1, (256, 2))
    update_D(X, Zb, net_D_q, net_G_q, trainer_D)
```

We compare the critic with the analytic λ, a difference of two Gaussian log densities, at two hundred real and two hundred generated samples, through the correlation and the fraction R^2 of the variance of λ that the critic explains; a constant critic has $R^2 = 0$.

```
def gaussian_logpdf(x, mu, Sigma):
    diff = x - mu
    sol = torch.linalg.solve(Sigma, diff.T).T
    return -0.5 * ((diff * sol).sum(1)
                   + torch.logdet(2 * torch.pi * Sigma))

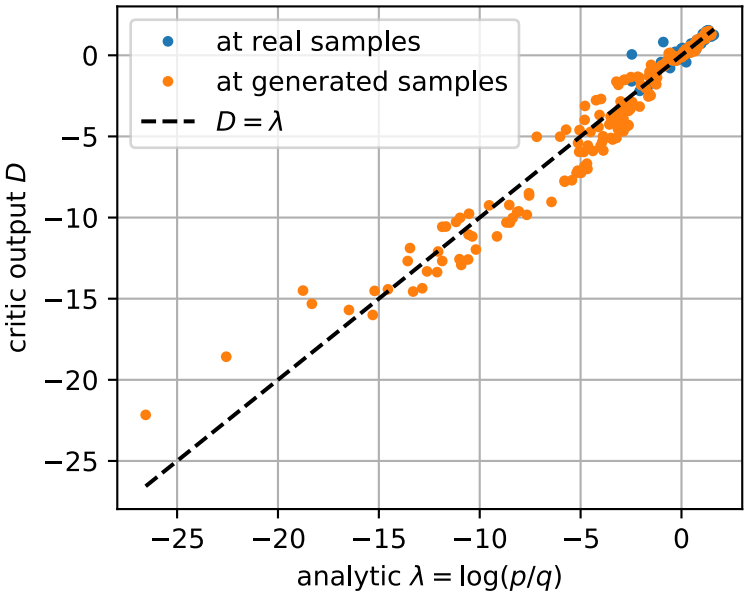
mu_q, Sigma_q = generator_gaussian(net_G_q)
with torch.no_grad():
    xs = torch.cat([data[:200], net_G_q(torch.normal(0, 1, (200, 2)))])
    lam = (gaussian_logpdf(xs, b, A.T @ A)
           - gaussian_logpdf(xs, mu_q, Sigma_q))
    D_out = net_D_q(xs).squeeze()
r2 = 1 - ((D_out - lam) ** 2).mean() / ((lam - lam.mean()) ** 2).mean()
print(f'correlation {torch.corrcoef(torch.stack([lam, D_out]))[0, 1]:.3f}, '
      f'R^2 {r2:.3f}')
```

```
correlation 0.981, R^2 0.962
```

In the scatter plot a perfect critic lies on the dashed diagonal.

```
lims = [float(lam.min()), float(lam.max())]
d2l.plot([lam[:200], lam[200:], lims], [D_out[:200], D_out[200:], lims],
```

```
        r'analytic $\lambda = \log(p/q)$', 'critic output $D$',
        legend=['at real samples', 'at generated samples', r'$D = \lambda$'],
        fmts=('C0.', 'C1.', 'k--'), figsize=(4.2, 3.4))
```



A network trained only to classify has learned the log density ratio: the correlation exceeds 0.97 and the critic explains more than 90 percent of the variance of λ. The points leave the diagonal mainly at the extreme values of λ, at generated samples where p is nearly zero. The objective constrains the critic only where the mixture m has appreciable mass, so the estimate is least accurate where the ratio is most extreme.

16.1.5.4 Saturating and Non-Saturating Losses Compared

Testing the two weights of Equation 16.6 needs a start at which they differ. The initial generator above emits samples about one standard deviation from the data mean, which the critic never rejects confidently enough to drive w_{sat} toward zero. We therefore move the generator's bias far from the data. The saturating update minimizes $\mathbb{E}_z[\log \sigma(-D(G(z)))]$, the minimax game taken literally.

```
def update_G_saturating(Z, net_D, net_G, trainer_G):
    trainer_G.zero_grad()
    fake_Y = net_D(net_G(Z))
    # log sigma(-D) is minus the cross-entropy under the label zero
    loss_G = -F.binary_cross_entropy_with_logits(
        fake_Y, torch.zeros_like(fake_Y), reduction='sum')
    loss_G.backward()
    trainer_G.step()
    return loss_G
```

Two arms start from the same initialization, see the same minibatches and latent draws,

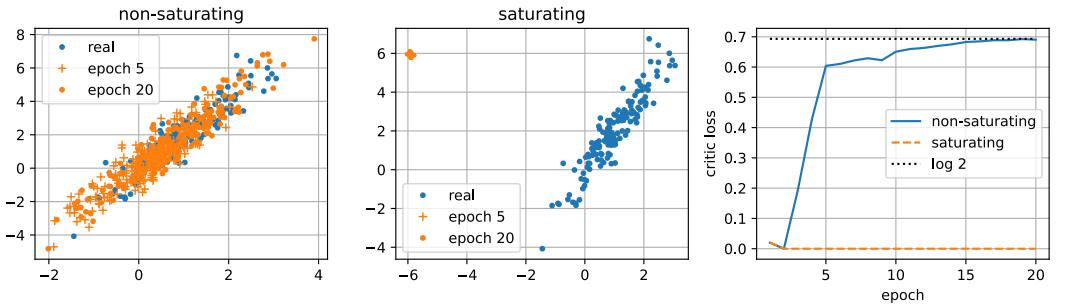
and differ only in the generator loss. After every epoch the callback `on_epoch` records the generator's samples at one fixed set of latent draws.

```
torch.manual_seed(3)
Z_show = torch.randn(200, 2)  # one latent sample for every snapshot
runs = {}
for name, step in [('non-saturating', update_G),
                   ('saturating', update_G_saturating)]:
    net_G_ab, net_D_ab = make_nets()
    with torch.no_grad():
        net_G_ab[0].bias += torch.tensor([-6.0, 6.0])
    snaps = {}  # epoch: generated samples
    hist = train_gan(net_D_ab, net_G_ab, data_iter, num_epochs=20,
                     latent_dim=2, lr_D=0.05, lr_G=0.01, step_G=step,
                     seed=1,  # the same minibatches and latent draws
                     on_epoch=lambda epoch: snaps.update(
                         {epoch: net_G_ab(Z_show).detach()}))
    runs[name] = (hist, snaps)
    mu_q, Sigma_q = generator_gaussian(net_G_ab)
    print(f'{name}: final KL(q || p) = '
          f'{kl_gaussians(mu_q, Sigma_q, mu_p, Sigma_p):.2f} nats')
```

```
non-saturating: final KL(q || p) = 0.02 nats
saturating: final KL(q || p) = 337.75 nats
```

The first two panels show each arm's samples after five and twenty epochs; the third shows both critics' losses.

```
fig, axes = d2l.plt.subplots(1, 3, figsize=(10.5, 3.2))
for ax, (name, (hist, snaps)) in zip(axes[:2], runs.items()):
    d2l.plot([data[:150, 0], snaps[5][:, 0], snaps[20][:, 0]],
             [data[:150, 1], snaps[5][:, 1], snaps[20][:, 1]],
             legend=['real', 'epoch 5', 'epoch 20'],
             fmts=('C0.', 'C1+', 'C1.'), axes=ax)
    ax.set_title(name)
epochs = list(range(1, 21))
d2l.plot(epochs, [runs['non-saturating'][0][:, 0],
                  runs['saturating'][0][:, 0], [math.log(2)] * 20],
         'epoch', 'critic loss', legend=['non-saturating', 'saturating',
                                         'log 2'],
         fmts=('-', '--', 'k:'), axes=axes[2])
fig.tight_layout()
```



The saturating generator does not move. Its critic's loss drops to zero within the first epochs, so $\sigma(D(x')) \approx 0$ for every generated sample, and the run ends where it started, more than three hundred nats from the data. The non-saturating generator crosses to the data within the first few epochs: by epoch five its samples lie in the data region, though not yet centered on it. By epoch twenty they fill the ellipse, $D_{\mathrm{KL}}(q \,\|\, p)$ is well below one nat, and its critic's loss is back at $\log 2$.

The non-saturating weight amplifies the factor $\nabla_\theta D$ but cannot create it. Here the critic's score surface slopes toward the data even far from them; an optimal critic on separated supports offers no slope, and no choice of weight helps.

16.1.6 Summary

A GAN replaces the likelihood with a learned comparison: a critic learns to tell real from generated samples, and the generator learns to defeat it. With the log loss, the optimal critic's logit is the log density ratio λ, and the value of the game is $2\,\mathrm{JS}(p, q) - 2\log 2$, where JS is the information a sample carries about its source. The non-saturating generator loss keeps the fixed point $q = p$ and, unlike the saturating one, keeps learning from rejected samples. Two perceptrons trained this way draw recognizable garments, though the critic still wins after twenty epochs; on a Gaussian, the fit and the learned ratio check out in nats. Two limits remain: the value is constant on separated supports, and the generator's divergence charges little for a dropped mode. Section 16.2 changes the loss and the critic class to obtain discrepancies that keep a gradient when the supports separate.

16.1.7 Exercises

1. **Likelihood versus samples.** Replace the weights of the noise mixture Equation 16.1 by ϵ and $1 - \epsilon$.
    1. Write the shortfall $\mathbb{E}_{p_{\mathrm{data}}}[\log p_{\mathrm{data}} - \log \tilde{p}]$ exactly as an expectation of a function of $p_{\mathrm{noise}}/p_{\mathrm{data}}$, and show that it is at most $\log(1/\epsilon)$. When is the bound nearly attained, and is it for natural images with p_{noise} uniform on the pixel cube?
    2. Convert the bound for $\epsilon = 0.01$ into bits per dimension (the negative log-likelihood divided by $d_x \log 2$) for 32×32 color images. Would a results table rounded to two decimals show it?
2. **Excess loss and the additive constant.** Let $\eta = p/(p+q)$ and let $D_{\mathrm{b}}(a \,\|\, u)$ be the Kullback–Leibler divergence between Bernoulli distributions with success probabilities a and u.

   1. Show that every critic satisfies $V(D^\star) - V(D) = 2\,\mathbb{E}_m\big[D_{\mathrm{b}}(\eta \,\|\, \sigma(D))\big]$, so that $D^\star$ is the unique maximizer of Equation 16.2 up to m-null sets.
   2. Conclude that $V(D^\star + b) < V(D^\star)$ for every constant $b \neq 0$.
   3. Expand to second order in b to show $V(D^\star + b) - V(D^\star) = -\frac{b^2}{2} \int \frac{pq}{p+q}\, dx + O(b^3)$, and that the coefficient lies in $[0, \frac{1}{2}]$. When the critic separates the samples easily, how sharply do finite samples fix its additive constant?

3. **Unequal priors.** Let the labeling problem draw $y = 1$ with probability α, so that the critic maximizes $V_\alpha(D) = \alpha\,\mathbb{E}_p[\log \sigma(D)] + (1-\alpha)\,\mathbb{E}_q[\log \sigma(-D)]$.

   1. Show that $D^\star_\alpha = \lambda + \log \frac{\alpha}{1-\alpha}$, and that $\max_D V_\alpha + H_b(\alpha)$ is the mutual information between a sample and its label, which lies between 0 and $H_b(\alpha)$.
   2. If `update_D` received twice as many real as generated samples, which α would it implement? How would the best response and w_{ns} change, and would $q = p$ remain a stationary point of the generator update?

4. **Supports of measure zero.**

   1. A generator $G(z) = zv + b_q$ with a scalar latent z places its samples on a line. Show that $\mathrm{JS}(p, q) = \log 2$ against the data distribution $p = \mathcal{N}(b, A^\top A)$ for every v and b_q.
   2. Show that $\mathrm{JS}(\hat{p}, q) = \log 2$ for the empirical distribution $\hat{p}$ of the thousand training points and every q with a density, $q = p$ included.
   3. Yet training on $\hat{p}$ fits q to a few hundredths of a nat. For a κ-Lipschitz critic, such as that of `make_nets`, use the Kantorovich–Rubinstein duality (Section 37.4.2) to show that replacing p by $\hat{p}$ in Equation 16.2 changes $V(D)$ by at most $\kappa\, W_1(\hat{p}, p)$. Why does this make the restricted game informative, and what happens as κ grows?

5. code **A dead critic.** ⊠ Give the critic of `make_nets` tanh units and hidden layers of 5 and 3 units.

   1. Rerun the non-saturating arm of the far-start comparison of Section 16.1.5.4 for five seeds, and record the final $D_{\mathrm{KL}}(q \,\|\, p)$ and critic loss.
   2. For a run that stalls, inspect the hidden activations and $\|\nabla_x D\|$ at real and generated samples. How can Adam at the critic's rate of 0.05 drive the units there while the critic rejects a distant generator, and why does the generator then get no direction?
   3. Show that a constant critic has the loss $\log 2$ at best. Which of the two readings of a critic loss of $\log 2$, a best response to $q = p$ or a critic that cannot separate the samples, fits the stalled run, and which measurement of this section tells them apart?

6. code **Two time scales.** ⊠ Rerun the Gaussian training of Section 16.1.5.1 with (`lr_D`, `lr_G`) set to $(0.05, 0.005)$, $(0.005, 0.005)$ and $(0.005, 0.05)$, five seeds each. Report the median and range of the final $D_{\mathrm{KL}}(q \,\|\, p)$. Is the spread across seeds small compared with the differences between settings? Against which critic does the generator optimize when it adapts faster than the critic?

7. code **Classifier two-sample test.** ⊠ A classifier two-sample test (Lopez-Paz and Oquab 2017) asks whether a classifier separates fresh real from fresh generated samples better than chance.

1. Show that the Bayes classifier has the accuracy $\frac{1}{2}(1 + \mathrm{TV}(p, q))$ (Section 37.3.3), and use Pinsker's inequality with the printed $D_{\mathrm{KL}}(q \,\|\, p)$ to bound the accuracy any classifier can reach against the trained Gaussian generator `net_G`.
2. Train the fresh critic `make_nets()[1]` against the frozen `net_G` and measure its accuracy on 2,000 held-out samples per class. How close does it come to the bound?
3. With accuracy approximately $\mathcal{N}(\frac{1}{2}, 1/(4n))$ under $q = p$, how many test points n would reject $q = p$ at the 5 percent level? What does a test with fewer points say about the generator?

8. code **Class coverage of the garment generator.** ▯ Thirty samples cannot show whether the generator of Section 16.1.4.2 draws each kind of clothing as often as the data do.

   1. Train the LeNet of Section 6.6 on Fashion-MNIST and label 1,000 generated samples with it. Compare the class histogram with the uniform one of the data. Which classes are over- or under-represented? Look at the samples of the rarest class.
   2. Repeat for generators trained for 5 and for 40 epochs. Does longer training even out the histogram?
   3. The classifier assigns every sample to some class, even one that resembles none. How does this bias the histogram? Relate what you find to mode dropping (Section 16.1.3.3).

9. **Generator weights and their divergences.** ▯▯ A generator loss $\mathbb{E}_z[g(D(G(z)))]$ with the critic held fixed has the gradient $-\mathbb{E}_z[w(D)\,\nabla_\theta D(G(z))]$ with $w = -g'$. Let the critic be the best response $D = \lambda$ to the current generator q_{θ_0}, held fixed, and write $\rho = e^D = p/q_{\theta_0}$.

   1. Show that the saturating and non-saturating losses sum to $D_{\mathrm{KL}}(q \,\|\, p)$ at $D = \lambda$, with weight $w = 1$.
   2. For the f-divergence $D_f(p \,\|\, q_\theta) = \mathbb{E}_{q_\theta}[f(p/q_\theta)]$ (Section 37.3.1.2), show that $\nabla_\theta D_f\big|_{\theta_0} = -\mathbb{E}_z[\rho^2 f''(\rho)\,\nabla_\theta D(G(z))]$. Hint: the product rule gives $\int \nabla_\theta q_\theta\, h$ with $h = f(\rho) - \rho f'(\rho)$. A weight w thus selects the f with $\rho^2 f''(\rho) = w(\log \rho)$.
   3. Identify the divergences selected by w_{sat}, by $w = 1$, by $w(t) = e^t$ (Goodfellow 2016), and by w_{ns} (Theorem 2.5 of Arjovsky and Bottou (2017)). Why may the critic be held fixed when differentiating the maximum over D?
   4. Show that the divergence of w_{ns}, $D_{\mathrm{KL}}(q \,\|\, p) - 2\,\mathrm{JS}(p, q)$, equals $2\,D_{\mathrm{KL}}(m \,\|\, p)$. Arjovsky and Bottou (2017) read its negative Jensen–Shannon term as pushing p and q apart. Does that reading hold?

   Adapted from Stefano Ermon and Aditya Grover, CS236: Deep Generative Models, Homework 3, Problem 2.3.

10. [extended] **Finite samples and critic capacity.** ▯▯ Once the number of samples exceeds the number of critic parameters, up to logarithmic factors, a critic's value on the samples is close to its value on the populations (Arora et al. 2017). Test this with the generator set to $zA + b$, so that $q = p$.

    1. Train critics with two hidden layers of width 4, 16, 128 and 1024 on $n \in \{100, 1000, 10000\}$ points from p (Adam at 10^{-3}, 4,000 steps of 256). Plot the gap between the loss on the training points and on fresh draws against the number of parameters, one curve per n. Where does the 337-parameter critic of `make_nets`

lie?

2. The garment critic of Section 16.1.4.2 has about 530,000 parameters and sees the same 60,000 images every epoch. What can its loss below $\log 2$ no longer tell apart? Propose a measurement that separates the two causes.

16.2 Adversarial Objectives and Divergences

The garments of Section 16.1 were trained with one loss, and after twenty epochs the critic still told real garments from generated ones with some confidence. Open almost any GAN codebase and its training script offers a menu of losses: non-saturating, least squares, hinge, Wasserstein. Practitioners switch between them the way they switch optimizers, keeping whichever trains. Section 16.1 analyzed only the first. Its critic learned the log ratio of the two densities, and its game measured the Jensen–Shannon divergence, which stops changing once the generated samples no longer overlap the data; an optimal critic then gives the generator no direction. Real and generated images typically lie on nearly disjoint sets, so a critic that stays ahead, as on the garments, comes close to that case. What do the other losses compute, and does any of them keep a gradient when the supports separate?

Every entry of the menu changes one of two things: the loss that trains the critic, or the set of functions the critic may be. We work out what each choice computes, check the predictions on a two-dimensional Gaussian mixture whose densities we know, and train one generator under five losses.

```
from torch.nn.functional import relu, softplus
```

16.2.1 A Common Template and Two Choices

16.2.1.1 The Template

The value function of Section 16.1 scores a real sample $x \sim p$ through $\log \sigma(D(x))$ and a generated sample $x' \sim q$ through $\log \sigma(-D(x'))$, where D is the critic's logit. We keep that section's notation: the mixture $m = (p+q)/2$, the density ratio $\rho = p/q$, and the optimal logit $\lambda = \log \rho$. Replacing the two log-probabilities by general scoring functions, ℓ_p for real samples and ℓ_q for generated ones, gives the template

$$d(p,q) \;=\; \sup_{T \in \mathcal{T}} \Big\{ \mathbb{E}_{x \sim p}\big[\ell_p(T(x))\big] - \mathbb{E}_{x' \sim q}\big[\ell_q(T(x'))\big] \Big\}, \tag{16.9}$$

in which the critic T ranges over a class $\mathcal{T}$. The game of Section 16.1 is the case $\ell_p(t) = \log \sigma(t)$ and $\ell_q(t) = -\log \sigma(-t)$ with $\mathcal{T}$ unrestricted.

The first family keeps the critic unconstrained and varies the scores. The supremum can then be taken separately at each x, so the result depends on p and q only through $\rho(x)$; these objectives are f-divergences. The second family uses linear scores, $\ell_p = \ell_q = \mathrm{id}$, and restricts $\mathcal{T}$ to a ball of functions, which couples the critic's values across points. A kernel ball gives the maximum mean discrepancy, and the ball of 1-Lipschitz functions the Wasserstein-1 distance (Figure 16.3).

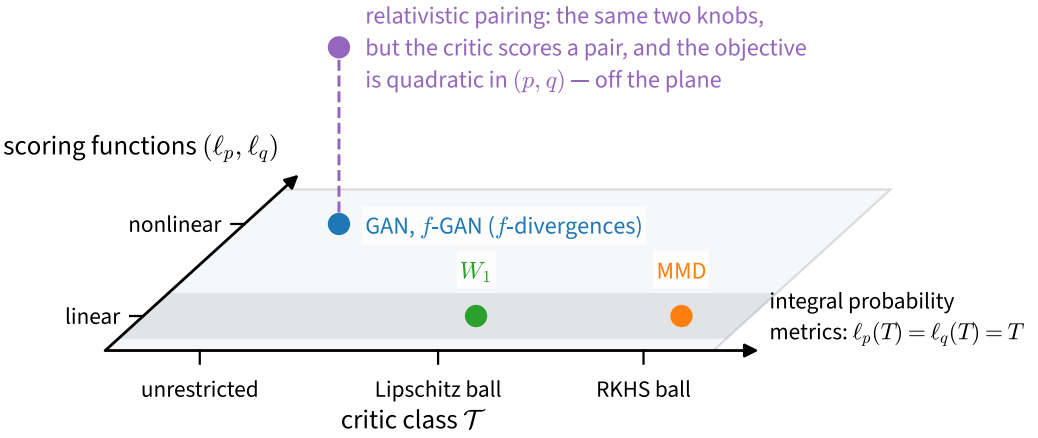


Figure 16.3: Two choices define the classical adversarial objectives: the scoring functions and the critic class. Nonlinear scores with an unconstrained critic yield f-divergences, including the original GAN. Linear scores with a constrained critic yield integral probability metrics: MMD for a kernel ball, Wasserstein-1 for a Lipschitz ball. The relativistic objective of Section 16.3 scores pairs of samples.

16.2.1.2 Convexity in the Generator's Distribution

For a fixed critic, the expression in braces is affine in the pair (p, q): linear in p and linear in q. Every objective of the template is therefore a supremum of affine functionals, hence jointly convex in (p, q). In particular, for fixed data p it is convex in q, and in the space of distributions the generator's problem has no spurious local minima. Convexity in q does not make the objective convex in the network parameters that represent q (Exercise 1).

16.2.2 Classification Losses and Their Divergences

16.2.2.1 Two Outcomes by Hand

Take a coin. The data put probabilities $p = (\frac{3}{4}, \frac{1}{4})$ on heads and tails, and the generator $q = (\frac{1}{4}, \frac{3}{4})$. Each toss comes from the data or from the generator with probability one half, and the critic, shown the outcome, must say which source produced it. Seeing heads, the source is real with probability $\eta = p/(p+q) = \frac{3}{4}$; seeing tails, with probability $\frac{1}{4}$. A critic that ignores the outcome can only report one half. How much looking helps depends on the loss:

- Under the log loss, the best report is the posterior itself. Its expected loss is the binary entropy $H_b(\frac{3}{4}) = 0.562$ nats, against $\log 2 = 0.693$ nats for the blind report. Looking saves 0.131 nats, which is $\mathrm{JS}(p, q)$.
- Under the squared error of a reported probability, the best report is again the posterior, with expected loss $\frac{3}{4} \cdot \frac{1}{4} = \frac{3}{16}$ against $\frac{1}{4}$ for the blind report. Looking saves $\frac{1}{16}$.
- Under the hinge loss on a score t, the best scores are $t = +1$ on heads and $t = -1$ on tails, with expected loss $2 \cdot \frac{1}{4} = \frac{1}{2}$ against 1 at $t = 0$. Looking saves $\frac{1}{2}$, which is the total variation distance $\frac{1}{2} \sum |p - q|$.

The three losses assign three numbers to one pair of distributions, all from the same pos-

terior, and each saving vanishes when $p = q$. We now show that every such saving is an f-divergence.

16.2.2.2 The Bayes-Risk Gap

Let ℓ be a payoff on critic outputs, so that $-\ell$ is a loss. A real sample earns $\ell(D(x))$ and a generated sample earns $\ell(-D(x'))$:

$$V_\ell(D) \;=\; \mathbb{E}_{x\sim p}\big[\ell(D(x))\big] + \mathbb{E}_{x'\sim q}\big[\ell(-D(x'))\big]. \tag{16.10}$$

The choice $\ell = \log\sigma$ recovers the log-loss game. The critic's signed score, $D(x)$ on a real sample and $-D(x')$ on a generated one, is its *margin*, and $-\ell$ is a *margin loss*. As in Section 16.1, the maximization decouples across points. At each x the integrand is $p\,\ell(t) + q\,\ell(-t)$ with $t = D(x)$, which is $2m(x)$ times an average under the posterior $\eta(x) = p(x)/(p(x)+q(x))$. The smallest expected loss a critic can reach at posterior η is the *conditional Bayes risk*

$$L(\eta) \;=\; \inf_t \Big\{\eta\,(-\ell(t)) + (1-\eta)\,(-\ell(-t))\Big\}. \tag{16.11}$$

As an infimum of functions linear in η, L is concave for every ℓ. A critic that ignores its input already reaches the risk $L(\frac{1}{2})$. Subtracting this baseline leaves what observing the sample contributes, the *Bayes-risk gap*

$$\Delta_\ell(p,q) \;=\; L(\tfrac{1}{2}) - \mathbb{E}_{x\sim m}\big[L(\eta(x))\big]. \tag{16.12}$$

It fixes the value of the game, $\sup_D V_\ell = 2\Delta_\ell - 2L(\frac{1}{2})$, and it is nonnegative by Jensen's inequality, because L is concave and $\mathbb{E}_m[\eta] = \frac{1}{2}$. For the log loss, L is the binary entropy, and the gap is the Jensen–Shannon divergence. The three savings of the coin are three Bayes-risk gaps.

16.2.2.3 Bayes-Risk Gaps as f-Divergences

An f-divergence averages a convex function of the density ratio under q, $D_f(p\,\|\,q) = \mathbb{E}_{x\sim q}[f(\rho(x))]$, where f is convex with $f(1) = 0$. Section 37.3.1.2 proves that $D_f \geq 0$ and fixes the conventions where one density vanishes. We call f the *divergence function*; the f-GAN literature calls it the generator, a word this chapter keeps for the network.

Proposition. *For any loss with concave Bayes risk L,*

$$\Delta_\ell(p,q) \;=\; D_f(p\,\|\,q) \qquad \text{with} \qquad f(u) \;=\; L(\tfrac{1}{2}) - \frac{u+1}{2}\,L\!\left(\frac{u}{u+1}\right),$$

and this f is convex with $f(1) = 0$.

Proof. Substitute $u = \rho = p/q$, so that $m = q\,(u+1)/2$ and $\eta = u/(u+1)$. Then

$$\mathbb{E}_{x\sim m}\big[L(\eta(x))\big] = \int q\,\frac{u+1}{2}\,L\!\left(\frac{u}{u+1}\right),$$

and since $\int q = 1$ the constant $L(\frac{1}{2})$ can be moved inside the integral, giving $\Delta_\ell = \int q\, f(u) = \mathbb{E}_q[f(\rho)]$. For convexity, write the concave L as the lower envelope of its tangent lines, $L(v) = \inf_j\{\alpha_j v + \beta_j\}$. Then

$$(u+1)\, L\!\left(\frac{u}{u+1}\right) = \inf_j \{\alpha_j u + \beta_j (u+1)\}$$

is an infimum of affine functions of u, hence concave, so f is convex; and $f(1) = L(\frac{1}{2}) - L(\frac{1}{2}) = 0$. ■

Thus every classification loss with a free critic evaluates the f-divergence built from its Bayes risk. The optimal critic is still a fixed function of the posterior $\eta(x)$, and hence of $\lambda(x)$, but each loss reports it through its own *link*: the logistic critic reports λ itself, the hinge critic only its sign.

16.2.2.4 Divergences of Common Losses

Table 16.2 evaluates Equation 16.12 for four common losses, with the binary entropy $H_b(\eta)$ and the total variation distance $\mathrm{TV}(p,q) = \frac{1}{2}\int |p-q|$. The square row scores a probability against zero–one targets. The squared hinge pays $\max(0, 1-t)^2$ on a margin t; its optimum $t = 2\eta - 1$ lies inside the margin, where it agrees with the square loss on ± 1 targets, so its gap is four times the square row's.

Table 16.2: Classification losses in the game of a single critic: the conditional Bayes risk, its value at the prior, the Bayes-risk gap Δ_ℓ (the value of the game is $2\Delta_\ell - 2L(\frac{1}{2})$), and the optimal critic as a function of the log ratio.

loss	Bayes risk $L(\eta)$	$L(\frac{1}{2})$	gap $\Delta_\ell(p,q)$	optimal critic
logistic	$H_b(\eta)$	$\log 2$	$\mathrm{JS}(p,q)$	λ
square	$\eta(1-\eta)$	$1/4$	$\frac{1}{8}\int \frac{(p-q)^2}{p+q}$	$\sigma(\lambda)$
squared hinge	$4\eta(1-\eta)$	1	$\frac{1}{2}\int \frac{(p-q)^2}{p+q}$	$\tanh(\lambda/2)$
hinge	$2\min(\eta, 1-\eta)$	1	$\mathrm{TV}(p,q)$	$\operatorname{sign}\lambda$

At the coin pair the four gaps are 0.131, $\frac{1}{16}$, $\frac{1}{4}$ and $\frac{1}{2}$. The square row is the critic loss of LSGAN (Mao et al. 2017), and its gap is a quarter of the bounded divergence $\chi^2(p \,\|\, m) = \frac{1}{2}\int (p-q)^2/(p+q)$. The hinge loss came to adversarial training from support vector machines through the Geometric GAN (Lim and Ye 2017); at $\eta = \frac{1}{2}$ every score in $[-1, 1]$ is optimal for it.

16.2.2.5 Critic Links

A nonnegative loss has a bounded gap, $\Delta_\ell \le L(\frac{1}{2})$, so the unbounded forward KL divergence that maximum likelihood minimizes is the gap of no such loss. Fenchel duality builds a game for any convex divergence function f. With the conjugate $f^*(t) = \sup_u\{ut - f(u)\}$, every critic T certifies the lower bound

$$D_f(p \,\|\, q) \;\ge\; \mathbb{E}_{x\sim p}[T(x)] - \mathbb{E}_{x'\sim q}[f^*(T(x'))], \tag{16.13}$$

which Section 37.3.4 proves from the Fenchel–Young inequality $f(u) \geq ut - f^*(t)$. Training a network T to maximize the right-hand side is the f-GAN construction (Nowozin et al. 2016); its Jensen–Shannon member is the original GAN (Exercise 3). The bound is attained by $T^\star = f'(\rho)$, once more a monotone link of the ratio, and an imperfect critic underestimates the divergence. For the forward KL, $f(u) = u \log u$ has the conjugate $f^*(t) = e^{t-1}$, and the optimal critic is $1 + \lambda$.

We can test the links point by point when both densities are known. The data p are a uniform mixture of three Gaussians in the plane. For q we set a second mixture by hand, with each mean moved by up to 0.3 and the components made wider, narrower and sheared, so that λ has a closed form at every point.

```
mu_p = np.array([[0.0, 2.4], [-2.1, -1.2], [2.1, -1.2]])     # data
A_p = np.stack([0.7 * np.eye(2)] * 3)
mu_q = np.array([[0.2, 2.6], [-2.3, -1.0], [2.0, -1.3]])     # hand-set q
A_q = np.stack([np.diag([0.8, 0.55]), np.diag([0.5, 0.6]),
                np.array([[0.65, 0.2], [-0.1, 0.7]])])

def mixture_logpdf(x, mus, As):
    """Log density of the uniform mixture of Gaussians N(mu_k, A_k^T A_k)."""
    comps = []
    for mu, A in zip(mus, As):
        Sigma, diff = A.T @ A, x - mu
        sol = np.linalg.solve(Sigma, diff.T).T
        comps.append(-0.5 * ((diff * sol).sum(1)
                             + np.log(np.linalg.det(2 * np.pi * Sigma))))
    return np.logaddexp.reduce(np.stack(comps) - np.log(len(mus)), axis=0)

def sample_mixture(n, mus, As, rng):
    k = rng.integers(0, len(mus), n)
    z = rng.standard_normal((n, 2))
    return np.einsum('ni,nij->nj', z, As[k]) + mus[k]
```

On a fixed grid, quadrature gives $\mathrm{JS}(p, q)$ for any mixture q of this form.

```
grid_1d = np.linspace(-7.0, 7.0, 281)
gx, gy = np.meshgrid(grid_1d, grid_1d)
grid_2d = np.stack([gx.ravel(), gy.ravel()], axis=1)
cell_area = (grid_1d[1] - grid_1d[0]) ** 2

def js_mixtures(mus_q, As_q, mus_p=mu_p, As_p=A_p):
    """JS(p, q) between two known mixtures, p the data by default, by
    quadrature on grid_2d."""
    logp = mixture_logpdf(grid_2d, mus_p, As_p)
    logq = mixture_logpdf(grid_2d, mus_q, As_q)
    p, q = np.exp(logp), np.exp(logq)
    logm = np.log(np.maximum((p + q) / 2, 1e-300))
    return 0.5 * (p * (logp - logm) + q * (logq - logm)).sum() * cell_area

print(f'JS(p, q) for the hand-set q: {js_mixtures(mu_q, A_q):.3f} nats')
```

```
JS(p, q) for the hand-set q: 0.033 nats
```

The hand-set q is a few hundredths of a nat from the data, a mismatch the size a generator meets late in training. The critic is a small leaky-ReLU network with the same initialization in both frameworks.

```
TRUNC_STD = 0.8796  # standard deviation of N(0, 1) truncated at +-2

def make_critic():
    """Leaky-ReLU MLP with LeCun-normal weights and zero biases."""
    D = nn.Sequential(nn.Linear(2, 64), nn.LeakyReLU(0.2),
                      nn.Linear(64, 64), nn.LeakyReLU(0.2), nn.Linear(64, 1))
    for layer in D[::2]:  # weight std 1/sqrt(fan_in) after truncation
        s = layer.in_features ** -0.5 / TRUNC_STD
        nn.init.trunc_normal_(layer.weight, std=s, a=-2 * s, b=2 * s)
        nn.init.zeros_(layer.bias)
    return D
```

Each loss is stored as a pair: the critic's loss on the scores of real and generated samples, and the generator's loss that Section 16.2.4.3 trains with.

```
losses = {  # name: (critic loss, generator loss), both on critic scores
    'logistic': (lambda r, f: (softplus(-r) + softplus(f)).mean(),
                 lambda f: softplus(-f).mean()),
    'least squares': (lambda r, f: ((r - 1) ** 2 + f ** 2).mean() / 2,
                      lambda f: ((f - 1) ** 2).mean() / 2),
    'hinge': (lambda r, f: (relu(1 - r) + relu(1 + f)).mean(),
              lambda f: -f.mean()),
    'squared hinge': (lambda r, f: (relu(1 - r) ** 2 + relu(1 + f) ** 2).mean(),
                      lambda f: -f.mean()),
}
```

The links describe the best response to a fixed q, so we train a fresh critic against the hand-set q under three of the losses. A fourth critic maximizes the forward-KL bound $\mathbb{E}_p[T] - \mathbb{E}_q[e^{T-1}]$.

```
to_t = lambda a: torch.tensor(a, dtype=torch.float32)

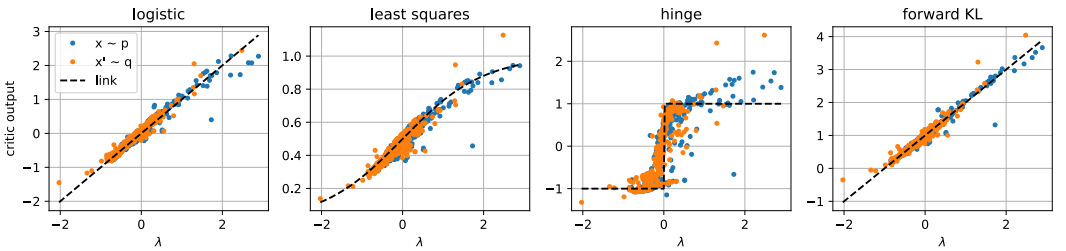
def best_response(critic_loss, seed=0, steps=3000, batch=512, lr=1e-3):
    """Fresh critic trained against the fixed q, as a NumPy function."""
    rng = np.random.default_rng(seed)
    torch.manual_seed(seed)
    D = make_critic()
    opt = torch.optim.Adam(D.parameters(), lr)
    for _ in range(steps):
        X = to_t(sample_mixture(batch, mu_p, A_p, rng))
        Y = to_t(sample_mixture(batch, mu_q, A_q, rng))
        opt.zero_grad()
        critic_loss(D(X), D(Y)).backward()
        opt.step()
    return lambda x: D(to_t(x)).squeeze(-1).detach().numpy()

kl_dual = lambda r, f: -(r.mean() - torch.exp(f - 1).mean())
```

```
critics = {name: best_response(losses[name][0])
           for name in ('logistic', 'least squares', 'hinge')}
critics['forward KL'] = best_response(kl_dual)
```

We evaluate each critic at 300 points drawn from p and 300 from q and plot its output against the analytic λ, with the predicted link as a dashed line.

```
rng = np.random.default_rng(7)
pts = np.concatenate([sample_mixture(300, mu_p, A_p, rng),
                      sample_mixture(300, mu_q, A_q, rng)])
lam = mixture_logpdf(pts, mu_p, A_p) - mixture_logpdf(pts, mu_q, A_q)
lam_line = np.linspace(lam.min(), lam.max(), 200)
links = {'logistic': lambda l: l, 'least squares': lambda l: 1 / (1 + np.exp(-l)),
         'hinge': np.sign, 'forward KL': lambda l: 1 + l}
fig, axes = d2l.plt.subplots(1, 4, figsize=(12, 3))
for ax, (name, link) in zip(axes, links.items()):
    out = critics[name](pts)
    d2l.plot([lam[:300], lam[300:], lam_line],
             [out[:300], out[300:], link(lam_line)], xlabel=r'$\lambda$',
             fmts=('.', '.', 'k--'), axes=ax,
             legend=['x ~ p', "x' ~ q", 'link'] if name == 'logistic' else None)
    ax.set_title(name)
axes[0].set_ylabel('critic output')
fig.tight_layout()
```



Each critic follows its own link of the same function: the identity, the sigmoid, a step at $\lambda = 0$, and the line $1 + \lambda$. All four drift from their links where $|\lambda|$ is large, because the objective constrains a critic only where samples land, and few land there. The hinge critic is the least determined; the training run below shows the consequence.

16.2.3 Integral Probability Metrics

The second family keeps linear scores and constrains the critic. The objective becomes

$$d_{\mathcal{F}}(p, q) \;=\; \sup_{T \in \mathcal{F}} \Big\{ \mathbb{E}_{x \sim p}\big[T(x)\big] - \mathbb{E}_{x' \sim q}\big[T(x')\big] \Big\}, \tag{16.14}$$

the largest difference in expectation that a test function in $\mathcal{F}$ can detect, an *integral probability metric* (Section 37.4.1). Its value no longer reduces to the pointwise ratio $\rho(x)$; it depends on which functions $\mathcal{F}$ contains.

16.2.3.1 Maximum Mean Discrepancy

Taking $\mathcal{F}$ to be the unit ball of a reproducing kernel Hilbert space with kernel k gives the maximum mean discrepancy (MMD). The supremum has a closed form (Gretton et al. 2012), and Section 37.4.1 derives its square as three kernel averages,

$$\mathrm{MMD}^2(p, q) \;=\; \mathbb{E}_{x,\tilde{x}\sim p}\big[k(x,\tilde{x})\big] + \mathbb{E}_{x',\tilde{x}'\sim q}\big[k(x',\tilde{x}')\big] - 2\,\mathbb{E}_{x\sim p,\,x'\sim q}\big[k(x,x')\big], \quad (16.15)$$

with the two samples in each average drawn independently: similarity within each distribution minus twice the similarity across them. For a fixed kernel, a generator can minimize MMD^2 directly, with no critic network, and the kernel alone decides what the discrepancy can see. The unbiased minibatch estimate averages each kernel matrix, leaving out the diagonal terms that compare a point with itself. The function below takes the three matrices, so it serves any kernel; Section 16.5 scores images with it.

```
def mmd2_unbiased(K_pp, K_qq, K_pq):
    """Unbiased MMD^2 from the kernel matrices within p, within q, and
    across; the diagonals (a point with itself) are left out."""
    m, n = len(K_pp), len(K_qq)
    return ((K_pp.sum() - K_pp.diagonal().sum()) / (m * (m - 1))
            + (K_qq.sum() - K_qq.diagonal().sum()) / (n * (n - 1))
            - 2 * K_pq.mean())
```

16.2.3.2 Wasserstein-1

Taking $\mathcal{F}$ to be the 1-Lipschitz functions gives the Wasserstein-1 distance W_1, the least average distance over which the mass of p must be moved to turn it into q (Section 37.4.2). On the line, W_1 is the area between the two cumulative distribution functions,

$$W_1(p, q) \;=\; \int_{-\infty}^{\infty} \big|F_p(t) - F_q(t)\big|\, dt,$$

so a translation by μ costs exactly $|\mu|$. In higher dimensions there is no closed form, so a WGAN trains a critic network and enforces the constraint approximately (Arjovsky et al. 2017), by weight clipping, a gradient penalty (Gulrajani et al. 2017) or spectral normalization (Miyato et al. 2018). Between two Gaussians the squared Wasserstein-2 distance also has a closed form, which Section 16.5 uses to score images.

16.2.3.3 Estimation from Samples

The two metrics differ in how well samples determine them. The unbiased MMD estimate converges at the rate $O(n^{-1/2})$ in any dimension (Gretton et al. 2012), whereas the empirical W_1 in dimension $d_x > 2$ converges at the rate n^{-1/d_x} (Weed and Bach 2019), so halving its error can take 2^{d_x} times as many samples. Density-ratio objectives are hardest to estimate where the ratio is extreme: at the optimal forward-KL critic, each term e^{T-1} of the minibatch average under q equals ρ and has variance $\chi^2(p \,\|\, q)$, which can be infinite even when the KL divergence is finite.

16.2.4 Gradients Far from and Near the Data

The two families disagree about the failure that closed Section 16.1. An f-divergence integrates a function of $\rho(x)$, which on disjoint supports is 0 or ∞ at every point, so nothing records how far apart the supports are: the divergence stays at its ceiling, and the generator's gradient vanishes. An integral probability metric averages one test function under both distributions, and if the test functions vary smoothly in space, its value changes as the supports move. Smoothness is the condition, not the family: total variation is the integral probability metric of the functions bounded by $\frac{1}{2}$ (Sriperumbudur et al. 2009), yet it saturates on disjoint supports, because that class contains functions as sharp as indicators.

16.2.4.1 Divergence against Separation

We can measure how much gradient each objective keeps as $p = \mathcal{N}(0, 1)$ and $q = \mathcal{N}(\mu, 1)$ move apart. All three can be evaluated without sampling error: Jensen–Shannon by quadrature, and $W_1 = |\mu|$ exactly. For the RBF kernel $k(x, x') = \exp(-(x - x')^2/(2s^2))$ with bandwidth s, the kernel averages in Equation 16.15 are Gaussian integrals, which give

$$\mathrm{MMD}^2(p, q) = \frac{2s}{\sqrt{s^2 + 2}} \left(1 - \exp\left(-\frac{\mu^2}{2(s^2 + 2)}\right)\right). \tag{16.16}$$

We plot each objective's slope with respect to μ, the gradient a generator that controls μ receives, on a log axis, with bandwidth $s = 4$.

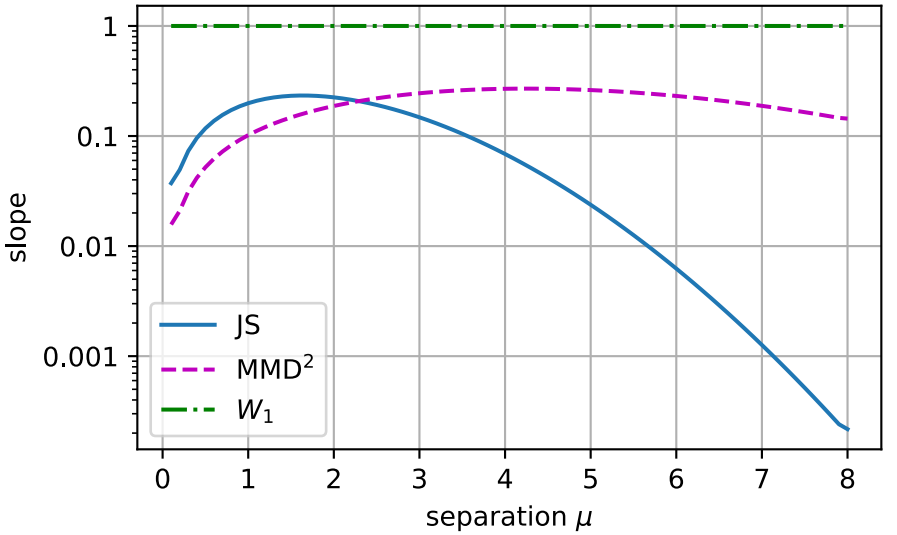
```
xs = np.linspace(-10.0, 18.0, 5601)          # fixed quadrature grid

def normal_pdf(x, mu, sigma=1.0):
    return np.exp(-(x - mu) ** 2 / (2 * sigma ** 2)) / np.sqrt(
        2 * np.pi * sigma ** 2)

def js_1d(p, q):
    m = (p + q) / 2
    return 0.5 * np.trapezoid(p * np.log(p / m) + q * np.log(q / m), xs)

s = 4.0                                       # RBF kernel bandwidth
mus = np.linspace(0.1, 8.0, 80)               # separations mu
js = np.array([js_1d(normal_pdf(xs, 0.0), normal_pdf(xs, mu)) for mu in mus])
mmd2 = 2 * s / np.sqrt(s ** 2 + 2) * (1 - np.exp(-mus ** 2
                                                 / (2 * (s ** 2 + 2))))
slopes = [np.gradient(js, mus), np.gradient(mmd2, mus),
          np.ones_like(mus)]                  # W1 = |mu| has slope one
for mu in (1.0, 7.0):
    print(f'mu = {mu:.0f}: JS slope {np.interp(mu, mus, slopes[0]):.4f}, '
          f'JS / log 2 = {np.interp(mu, mus, js) / np.log(2):.4f}')
d2l.plot(mus, slopes, xlabel=r'separation $\mu$', ylabel='slope',
         legend=['JS', 'MMD$^2$', '$W_1$'], yscale='log', figsize=(5, 3))
```

```
mu = 1: JS slope 0.1987, JS / log 2 = 0.1607
mu = 7: JS slope 0.0013, JS / log 2 = 0.9990
```



The Jensen–Shannon slope collapses once the densities stop overlapping. At $\mu = 7$ it is more than a hundred times smaller than at $\mu = 1$, and the divergence is within a tenth of a percent of $\log 2$, so the generator receives almost no signal about which way to move. The W_1 slope stays at one. The MMD^2 slope peaks at a separation set by the bandwidth and then decays like a Gaussian in μ, so the bandwidth sets the range over which MMD gives a gradient.

16.2.4.2 Optimal Critics on Separated Supports

Figure 16.4 shows why. The Jensen–Shannon critic λ is flat on each support, so moving q a little leaves every generated sample on the same level and carries no information about direction. The W_1 critic is a ramp of unit slope, so every generated sample sits on a slope toward the data, however far away. The MMD witness, a smooth difference of kernel bumps, slopes toward the data as long as the kernel reaches across the gap.

16.2.4.3 Five Losses on a Gaussian Mixture

Every divergence in Table 16.2 vanishes at $q = p$, so with overlapping supports every loss should drive a generator to the data. Near the end, though, the hinge critic reports only the sign of the remaining mismatch, the others also its size. We fit one generator to the three-mode mixture under five objectives: the non-saturating logistic loss of Section 16.1, least squares, hinge, the squared hinge, and MMD with a fixed kernel and no critic. The squared hinge keeps the hinge's margin and generator loss $-D$ but has the link $\tanh(\lambda/2)$, which carries magnitude, so comparing the two isolates the link. All five runs share the architecture, learning rates, step counts and start, with no separate tuning. A W_1 arm would need the Lipschitz penalty of Section 16.4.

The generator draws one of three components uniformly and applies that component's learned affine map to Gaussian noise, so q stays a three-component mixture whose divergence from p we can compute after every step. We set the start by hand: each mean lies one to two and a half units from a different mode, and each map is 0.4 times the identity, below the data's 0.7, so positions and scales remain to be learned.

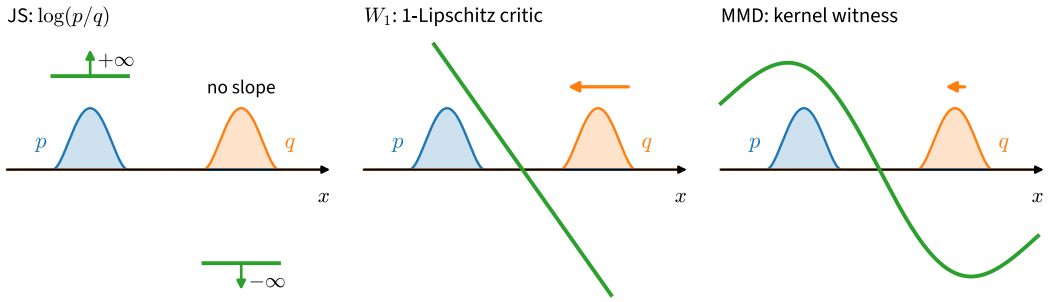


Figure 16.4: Optimal critics when the data density p (blue) and the generated density q (orange) have disjoint supports. Left: the Jensen–Shannon critic $\log(p/q)$, $+\infty$ on the support of p and $-\infty$ on that of q, flat on both. Middle: the W_1 critic, a ramp of slope one. Right: the MMD witness $\mathbb{E}_p[k(x,\cdot)]-\mathbb{E}_q[k(x,\cdot)]$, a smooth difference of kernel bumps (bandwidth half the separation, arbitrary scale). The orange arrows mark each critic's slope at q, the direction in which the generator moves.

```
mu_0 = np.array([[1.0, -1.0], [-4.3, -2.0], [-1.0, 3.8]])  # off the modes
A_0 = np.stack([0.4 * np.eye(2)] * 3)      # below the data's scale of 0.7
print(f'JS(p, q) at the start: {js_mixtures(mu_0, A_0):.3f} nats')
```

```
JS(p, q) at the start: 0.529 nats
```

The MMD arm minimizes the unbiased minibatch MMD^2 under the sum of three RBF kernels with bandwidths 0.5, 1 and 2, a kernel sensitive at several scales.

```
class MixtureGenerator(nn.Module):
    """x = mu_k + z A_k for a uniformly drawn component k."""
    def __init__(self, mu, A):
        super().__init__()
        self.mu = nn.Parameter(to_t(mu))
        self.A = nn.Parameter(to_t(A))

    def forward(self, z, k):
        return torch.einsum('ni,nij->nj', z, self.A[k]) + self.mu[k]

    def params_np(self):
        return (self.mu.detach().numpy().copy(),
                self.A.detach().numpy().copy())

def mmd2_loss(x, y, bandwidths=(0.5, 1.0, 2.0)):
    """Unbiased MMD^2 under the sum of three RBF kernels."""
    def kernel(a, b):
        sq_dist = torch.cdist(a, b) ** 2
        return sum(torch.exp(-sq_dist / (2 * s ** 2)) for s in bandwidths)
    return mmd2_unbiased(kernel(x, x), kernel(y, y), kernel(x, y))

losses['MMD'] = (None, mmd2_loss)   # no critic: MMD^2 to the real batch
```

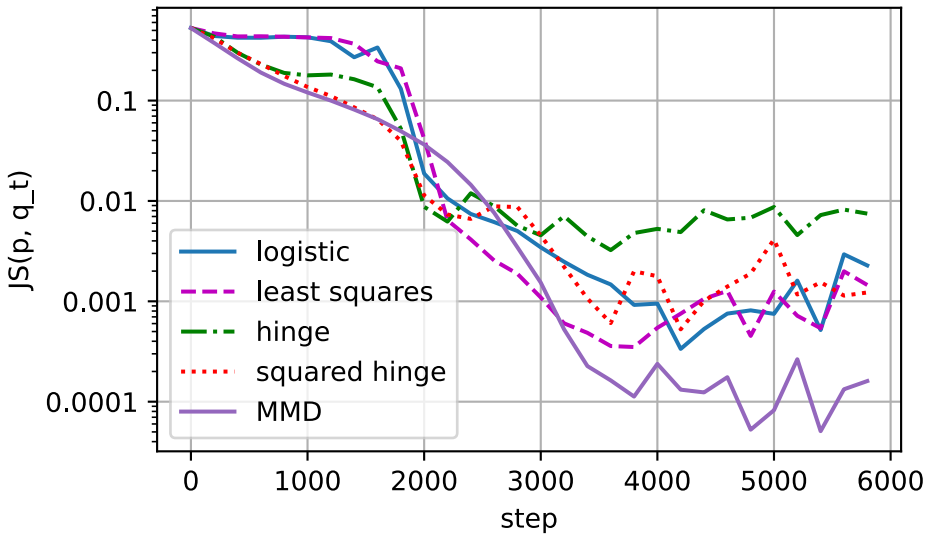
`train_toy` takes the two losses as arguments and alternates one critic step with one generator step. The MMD arm, stored with the critic loss `None`, takes only the generator step, on `mmd2_loss(X, fake)` instead of the critic's scores. Every 200 steps it records the exact $\mathrm{JS}(p, q_t)$.

```
def train_toy(critic_loss, gen_loss, seed=0, steps=6000, batch=256, lr=1e-3):
    rng = np.random.default_rng(seed)
    torch.manual_seed(seed)
    G, D = MixtureGenerator(mu_0, A_0), make_critic()
    opt_G = torch.optim.Adam(G.parameters(), lr)
    opt_D = torch.optim.Adam(D.parameters(), 2 * lr)
    trace = []
    for t in range(steps):
        X = to_t(sample_mixture(batch, mu_p, A_p, rng))
        z = to_t(rng.standard_normal((batch, 2)))
        k = torch.tensor(rng.integers(0, 3, batch))
        if critic_loss is not None:
            opt_D.zero_grad()
            critic_loss(D(X), D(G(z, k).detach())).backward()
            opt_D.step()
        opt_G.zero_grad()
        fake = G(z, k)
        loss = gen_loss(X, fake) if critic_loss is None else gen_loss(D(fake))
        loss.backward()
        opt_G.step()
        if t % 200 == 0:
            trace.append(js_mixtures(*G.params_np()))
    return np.array(trace)
```

The divergence fluctuates from step to step near the end of a run, so we report its median over the last 2,000 steps, and plot the traces.

```
runs = {name: train_toy(*pair) for name, pair in losses.items()}
for name, trace in runs.items():
    print(f'{name:14s} median JS(p, q) over the last 2,000 steps: '
          f'{np.median(trace[-2000 // 200:]):.4f} nats')
d2l.plot(np.arange(len(runs['logistic'])) * 200, list(runs.values()),
         xlabel='step', ylabel='JS(p, q_t)', yscale='log',
         legend=list(runs), figsize=(5, 3))
```

```
logistic       median JS(p, q) over the last 2,000 steps: 0.0008 nats
least squares  median JS(p, q) over the last 2,000 steps: 0.0009 nats
hinge          median JS(p, q) over the last 2,000 steps: 0.0070 nats
squared hinge  median JS(p, q) over the last 2,000 steps: 0.0013 nats
MMD            median JS(p, q) over the last 2,000 steps: 0.0001 nats
```



Every objective takes the generator from about half a nat to a few thousandths of a nat or less. The hinge ends highest, at about 0.005 nats, usually several times above the rest. The squared hinge, with the same margin and generator loss, ends below it, at the level of the logistic and least-squares games. MMD, which has no critic to keep up with, ends lowest. The logistic and least-squares runs also stay on their initial plateau longer than the other three.

The hinge stall comes from what its critic must track. Its target sign λ is a full-margin step however small the mismatch, and inside the margin the hinge risk pulls the critic toward it with slope $|1 - 2\eta| \approx |\lambda|/2$, which vanishes as q approaches p. The critic falls behind, and the generator keeps following a stale push of full size. The squared hinge's target $\tanh(\lambda/2)$ shrinks with the mismatch, and its quadratic risk pulls in proportion to the error, so its critic keeps up. All five losses share the minimizer $q = p$; what the critic reports near it decides how closely the generator approaches it (Exercise 6 tests the lag).

With overlapping supports every loss reaches the data, and studies that give each loss the same tuning budget find them comparable in final quality (Lucic et al. 2018). The choice among them matters less than two questions the losses leave open: what happens when the supports separate, and whether gradient training reaches the equilibrium. Before turning to those, the next section keeps the log loss and changes what the critic scores: a real sample against a generated one.

16.2.5 Summary

Every adversarial objective of this section is a supremum, over a critic class, of a difference of expected scores, so it is convex in the generator's distribution. With a free critic, a classification loss evaluates the f-divergence given by its Bayes-risk gap, and the trained critic reports a fixed link of the log ratio λ; Fenchel duality extends this to any divergence function. Constraining the critic instead gives integral probability metrics: MMD for a kernel ball, W_1 for the Lipschitz ball. When the supports separate, the ratio-based objectives saturate, while W_1 keeps unit slope and MMD keeps a slope over separations set by its bandwidth. When they overlap, every loss tested reaches the data, but the hinge, whose critic reports only a sign, stalls above the squared hinge and every other loss.

16.2.6 Exercises

1. code **Convexity and parameterization.** Every objective of the form Equation 16.9 is convex in q, but a generator moves q through its parameters. Use `normal_pdf`, `js_1d` and `xs`.
   1. Let $p = \mathcal{N}(0,1)$ and $q_0 = \mathcal{N}(8,1)$. Check with second differences that $\mathrm{JS}(p, q_t)$ is convex in t along the mixture path $q_t = (1-t)\,q_0 + t\,p$.
   2. Now let $p = 0.7\,\mathcal{N}(-4,1) + 0.3\,\mathcal{N}(4,1)$ and $q_\theta = \mathcal{N}(\theta, 1)$. Plot $\mathrm{JS}(p, q_\theta)$ for $\theta \in [-6, 6]$ and locate its local minima.
   3. Why do these minima not contradict the first part? Which property of the set $\{q_\theta\}$ does the convexity argument need?
2. code **Checking and extending the table.**
   1. The Brier score predicts a probability $\hat\eta$ and pays $(y - \hat\eta)^2$ against the source label $y \in \{0,1\}$. Show that its conditional Bayes risk Equation 16.11 is $\eta(1-\eta)$, attained at $\hat\eta = \eta$, and that its gap is $\frac{1}{8}\int (p-q)^2/(p+q)$. Confirm the factor $\frac{1}{8}$ on Dirichlet draws of pairs of distributions on five outcomes.
   2. The exponential loss pays e^{-t} on a margin t, and the zero–one loss pays one for a negative margin. Show that their Bayes risks are $2\sqrt{\eta(1-\eta)}$ and $\min(\eta, 1-\eta)$, that their gaps are the squared Hellinger distance $1 - \int \sqrt{pq}$ and $\frac{1}{2}\,\mathrm{TV}(p,q)$, and that the optimal exponential critic is $\lambda/2$. Evaluate both gaps at the coin pair.
3. **Jensen–Shannon conjugate.** Twice the Jensen–Shannon divergence has the divergence function $f(u) = u \log u - (u+1)\log \frac{u+1}{2}$.
   1. Derive $f^*(t) = -\log(2 - e^t)$, finite exactly for $t < \log 2$.
   2. Show that under $T = \log(2\sigma(D))$ the optimal critic $T^\star = f'(\rho)$ corresponds to $D^\star = \lambda$, and that the bound Equation 16.13 becomes $2\log 2 + V(D)$ with the value function V of Section 16.1.
   3. The activation $\log 2 - \mathrm{softplus}$ covers exactly the slopes $f'(u)$, and $-\mathrm{softplus}$ also maps into the domain of f^*. Which values of $T^\star$ does the second exclude, and what does it do to the bound whenever $p \neq q$?
4. code **MMD gradient range.**
   1. Differentiate Equation 16.16 with respect to μ and show that the slope is largest at $\mu^\star = \sqrt{s^2 + 2}$.
   2. For the three-bandwidth kernel of `mmd2_loss`, plot the slope of MMD^2 between the same two Gaussians for $\mu \in [0, 20]$, and find the separation beyond which it stays below 1% of its maximum. Compare with the single kernel $s = 4$.
5. **Lipschitz hinge critics.** ⊠ Spectrally normalized GANs score higher with the hinge loss than with the logistic loss (Miyato et al. 2018), so practice combines the loss of an f-divergence with a constrained critic. Write the hinge game as $V(D) = \mathbb{E}_p[-\max(0, 1-D)] + \mathbb{E}_q[-\max(0, 1+D)]$.
   1. Show that $V(D) = \mathbb{E}_p[D] - \mathbb{E}_q[D] - 2$ whenever $|D| \le 1$, and that clipping a critic to $[-1, 1]$ never lowers V. Conclude that the unconstrained hinge game is the integral probability metric of $\{T : \|T\|_\infty \le 1\}$, shifted by -2, and check its value against the table.
   2. Restrict the critic to κ-Lipschitz functions. Show that clipping preserves the Lipschitz constant, and evaluate the game for point masses at 0 and $\mu > 0$. Over

which separations does it give the generator a gradient, compared with W_1 and an RBF MMD with bandwidth s?

6. code **Plateaus and critic lag.** ▣ Use `train_toy`.
   1. Pair the logistic critic loss $-D$ and compare it with the non-saturating entry over three seeds by the first step at which $\mathrm{JS}(p, q_t) < 0.3$ nats. Explain the difference from the per-sample weights $\sigma(-D(x'))$ and 1 that the two generator losses give (Section 16.1).
   2. Test the explanation of the hinge residual with two controls over three seeds: quadruple the critic's batch, or give the critic five steps per generator step. Which control lowers the residual, and does the outcome point to minibatch noise or to lag?

7. code **Estimation and dimension.** ▣ Let $p = \mathcal{N}(\mathbf{0}, \mathbf{I}_{d_x})$ and $q = \mathcal{N}(\mu, \mathbf{I}_{d_x})$ with $\|\mu\| = 1$, so that $W_1(p, q) = 1$ in every dimension.
   1. Estimate W_1 from two samples of size n by an optimal assignment (`scipy.optimize.linear_sum_assignment`) for $d_x \in \{1, 2, 10, 50\}$ and $n \in \{100, 400, 1600\}$, and plot the relative error against n on log–log axes. Repeat with the unbiased MMD^2 estimate under an RBF kernel of bandwidth $\sqrt{d_x}$, whose population value is $2(s^2/(s^2+2))^{d_x/2}(1 - e^{-1/(2(s^2+2))})$, and compare the slopes with the quoted rates.
   2. Measure the empirical W_1 between two samples of size 256 from p itself in dimension 50. What does it say about a minibatch critic meant to resolve $W_1(p, q) = 1$?
   3. Evaluate the bound Equation 16.13 of `critics['forward KL']` by quadrature on `grid_2d`. A critic trained for 3,000 steps recovers roughly 85 to 95 percent of $D_{\mathrm{KL}}(p \,\|\, q) \approx 0.16$ nats. Then evaluate the bound on minibatches of 512. Why can a minibatch estimate flatter a critic that overshoots $1 + \lambda$ where q rarely samples?

8. **Divergences a loss can reach.** ▣▣ Every Bayes risk satisfies $L(\eta) = L(1-\eta)$, because Equation 16.10 scores real and generated samples with one function applied to D and $-D$, so every gap is a symmetric f-divergence. Corollary 3 of Nguyen et al. (2009) shows the converse, and this problem builds a nonnegative loss for it. Let f be convex on $(0, \infty)$ with $f(1) = 0$, $f(u) = uf(1/u)$, and $f(0) = \lim_{u \to 0} f(u) < \infty$, and define

$$L(\eta) = 2f(0) - 2(1-\eta)\, f\!\left(\frac{\eta}{1-\eta}\right), \qquad 0 < \eta < 1.$$

   1. Show that L is concave, symmetric about $\frac{1}{2}$, tends to 0 at both ends, and has Bayes-risk gap $D_f(p \,\|\, q)$.
   2. For differentiable L and $\hat{\eta} = \sigma(t)$, let a real sample cost $c(t) = L(\hat{\eta}) + (1 - \hat{\eta})L'(\hat{\eta})$. Show that a generated sample then costs $L(\hat{\eta}) - \hat{\eta}L'(\hat{\eta})$, that the conditional risk is minimized at $\hat{\eta} = \eta$ with value $L(\eta)$, and that $c \geq 0$.
   3. Apply the construction to the Jensen–Shannon divergence function and show that it returns the logistic loss.

9. [extended] **Separated initializations.** ▣▣ The five losses of Section 16.2.4.3 train only from overlapping supports. Let $p = \mathcal{N}(0, 1)$ and let the generator $q = \mathcal{N}(\mu, \tau^2)$ learn μ and $\log \tau$ from $\tau = 1$ and $\mu_0 \in \{2, 5, 10, 20\}$. Compare the logistic, least-squares and hinge games with a one-dimensional version of the critic, the hinge game

with a spectrally normalized critic, and MMD with RBF bandwidths $1, 4$ and the three-bandwidth kernel of `mmd2_loss`, at fixed optimizer settings and 5,000 generator steps. Tabulate, over three seeds, the steps until $\mathrm{JS}(p, q) < 0.01$, and explain every row from this section. Which runs stall, and where does a trained critic behave differently from the optimal critic of the analysis? Relate the rows to exercises 5 and 4 of this section.

16.3 Relativistic Objectives

In the previous two sections the critic scored one sample at a time. People rarely judge subtle differences that way. A taster who cannot say whether one glass holds the expensive wine can often say which of two glasses does, and a chess rating is built from who beat whom, not from an absolute score per player. The statistical model behind such ratings, the Bradley–Terry model, is the one Section 15.3 used to fit a reward model to human preferences. Jolicoeur-Martineau (2019) gave the GAN critic the same comparative task: show it one real and one generated sample, and reward it for ranking the real one higher.

This section asks what the change does to the game. Does the best ranking critic still estimate the density ratio of Section 16.1? Does a pair reveal more about which sample is real than a single sample does, and does that help when the two distributions are far apart? And how does comparison change the push that each generated sample receives? The answers say what pairing contributes to the modern recipe of the next two sections, and what it leaves for Section 16.4 to fix.

```
import itertools
```

16.3.1 Scoring Pairs

16.3.1.1 The Bradley–Terry Critic

We keep the notation of Section 16.1: data density p, generator density q, ratio $\rho = p/q$, log ratio $\lambda = \log \rho$, and a critic D whose value is a realness logit. Draw a real sample $x \sim p$ and a generated sample $x' \sim q$ independently, and score the pair by the difference of the two critic values:

$$\Phi(D) \;=\; \mathbb{E}_{x\sim p,\; x'\sim q}\big[\log \sigma\big(D(x) - D(x')\big)\big]. \tag{16.17}$$

The critic maximizes Φ, whereas the generator minimizes it. The quantity $\sigma(D(x)-D(x'))$ is the Bradley–Terry probability (Bradley and Terry 1952) that x beats x', so Φ is the log-likelihood of ranking real samples above generated ones, and the critic learns a ranking rather than a classification rule. Jolicoeur-Martineau called such a critic *relativistic*, because it scores a sample relative to a sample from the opposing distribution. The paired form used here is abbreviated RpGAN, for relativistic *paired*, and the divergence derived below inherits the subscript. Comparing one generated sample with several real ones gives the InfoNCE loss of contrastive learning (Exercise 6).

16.3.1.2 Shift Invariance and Bilinearity

Ranking ignores the level of the critic's scores, which classification does not (Figure 16.5).

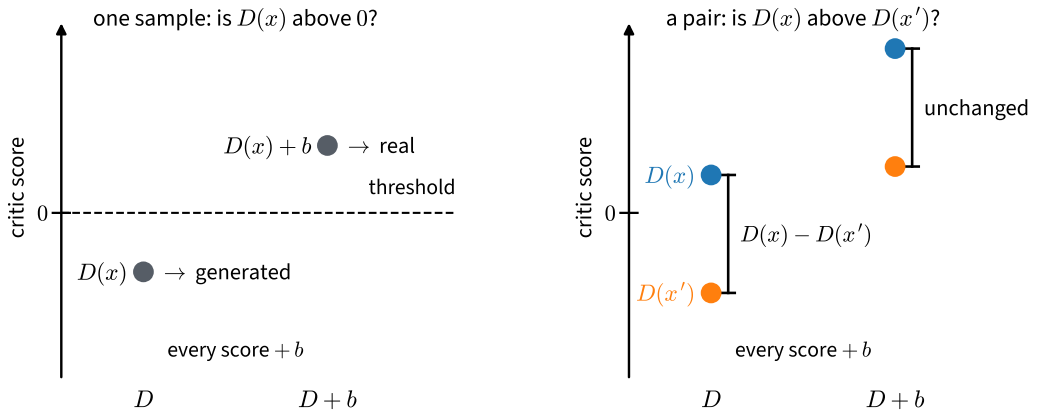


Figure 16.5: Scoring one sample against a threshold (left) versus scoring a pair by its difference (right). Adding a constant b to every score can change the classical verdict, which asks whether $D(x)$ clears a fixed level, but leaves the difference $D(x) - D(x')$ unchanged.

Formally, replacing D by $D + b$ for any constant b leaves every score difference, and therefore Φ, unchanged. The pairing game identifies the critic only up to an additive constant. The classical game of Section 16.1 determines this constant as well: shifting its optimal critic strictly decreases the objective (Exercise 2b of Section 16.1).

The objective also depends differently on the two distributions. The expectation in Equation 16.17 is taken under the product measure $p \otimes q$, so Φ is *bilinear* in (p, q): linear in each distribution while the other is held fixed, but not affine in the pair. The objectives of Section 16.2 were suprema of functionals affine in (p, q), which let the critic's problem decouple pointwise and made each value an f-divergence. Those results do not apply to the pairing game, so we compute its value directly.

16.3.2 The Value of the Pairing Game

16.3.2.1 The Optimal Critic

The inner maximization no longer decouples across points, because each critic value $D(t)$ interacts with every other through the differences. Concavity substitutes for decoupling.

Proposition. *Φ is concave in D, and if p and q share a common support it is maximized at $D^\star = \lambda$, uniquely up to an additive constant.*

Proof. The map $D \mapsto D(x) - D(x')$ is linear and $\log \sigma$ is concave, so the integrand of Equation 16.17 is concave in D for every pair, and the expectation preserves concavity; a stationary point is therefore a global maximum. Perturbing D at a single point t, that point enters Φ once as the real member of a pair and once, with opposite sign, as the generated member, and $\frac{d}{du} \log \sigma(u) = \sigma(-u)$ gives the functional derivative

$$\frac{\delta \Phi}{\delta D(t)} \;=\; p(t)\, \mathbb{E}_{x' \sim q}\big[\sigma(D(x') - D(t))\big] \;-\; q(t)\, \mathbb{E}_{x \sim p}\big[\sigma(D(t) - D(x))\big]. \tag{16.18}$$

At $D = \lambda$, we have $\sigma(\lambda(x') - \lambda(t)) = \rho(x')/(\rho(x') + \rho(t))$. The identity $q\rho = p$ converts

the first term of Equation 16.18 to $p(t)\int p(x')/(\rho(x')+\rho(t))\,dx'$. It converts the second to $q(t)\,\rho(t)\int p(x)/(\rho(t)+\rho(x))\,dx$, which is the same expression because $q(t)\rho(t) = p(t)$. The derivative therefore vanishes everywhere, so λ is a maximizer. Strict concavity of $\log\sigma$ in the score differences makes this maximizer unique up to an additive shift. ■

Ranking and classification therefore estimate the same log density ratio, since the log-loss critic of Section 16.1 also has the optimal logit λ. Replacing classification with ranking keeps the optimal critic and, as the theorem below confirms, the fixed point $q = p$. It changes the value and the shape of the objective away from that point.

16.3.2.2 Two Atoms by Hand

Before computing the value in general, take the smallest case. Let the space hold two atoms, 0 and 1, with the data distribution $p = (0.8, 0.2)$ and the generator distribution $q = (0.2, 0.8)$. A single sample at atom 0 is real with probability $p(0)/(p(0)+q(0)) = 0.8$, and a sample at atom 1 with probability 0.2. Now draw one real and one generated sample and look at them as an ordered pair (a_1, a_2). The table lists the probability of each pair when the real sample comes first, $P(a_1, a_2) = p(a_1)\,q(a_2)$, and when it comes second, $Q(a_1, a_2) = q(a_1)\,p(a_2)$:

pair (a_1, a_2)	$(0,0)$	$(1,1)$	$(0,1)$	$(1,0)$
P: real sample first	0.16	0.16	0.64	0.04
Q: real sample second	0.16	0.16	0.04	0.64

A pair of equal atoms is equally likely in both orders and says nothing about which member is real. The mixed pairs carry the information. When the pair reads $(0, 1)$, the real sample is the first with probability $0.64/(0.64 + 0.04) = 16/17 \approx 0.94$, more than the 0.8 that a single sample at atom 0 gives.

The optimal critic turns this into a value. Only the score difference $d = D(0) - D(1)$ matters, and the critic maximizes

$$\Phi = 0.32\log\sigma(0) + 0.64\log\sigma(d) + 0.04\log\sigma(-d).$$

Setting the derivative $0.64\,\sigma(-d) - 0.04\,\sigma(d)$ to zero gives $e^d = 16$, so $d = \log 16 = \lambda(0) - \lambda(1)$, as the proposition predicts. Substituting back and writing $H_b(t) = -t\log t - (1-t)\log(1-t)$ for the entropy of a coin with bias t,

$$\sup_D \Phi + \log 2 = 0.68(\log 2 - H_b(\tfrac{1}{17})) \approx 0.319 \text{ nats}.$$

Read this as information. The mixed pairs, which occur with probability 0.68, turn a fair coin about the real sample's position into a coin with bias $1/17$; the equal pairs leave it fair. For a single sample the same reading gives the Jensen–Shannon divergence, which Section 16.1 identified with the information one sample carries about its source: $\mathrm{JS}(p, q) = \log 2 - H_b(0.8) \approx 0.193$ nats. The pair carries 1.66 times as much. The cell below checks the three numbers; its `js` computes the divergence of two probability vectors from the entropy form $\mathrm{JS}(a, b) = H(\frac{1}{2}(a+b)) - \frac{1}{2}(H(a) + H(b))$ of Section 16.1.

```python
def entropy(v):
    v = v[v > 0]                        # the convention 0 log 0 = 0
    return -(v * np.log(v)).sum()

def js(a, b):
    return entropy((a + b) / 2) - (entropy(a) + entropy(b)) / 2

p, q = np.array([0.8, 0.2]), np.array([0.2, 0.8])
P, Q = np.outer(p, q), np.outer(q, p)  # P[i, j]: real sample at i, generated at j
lam = np.log(p / q)                    # the optimal critic: lam[0] - lam[1] = log 16
U = lam[:, None] - lam[None, :]        # U[i, j] = D(i) - D(j)
phi = (P * -np.logaddexp(0, -U)).sum() # Phi = E_P[log sigma(D(a_1) - D(a_2))]
print(f'JS(p, q) = {js(p, q):.3f}   JS(P, Q) = {js(P.ravel(), Q.ravel()):.3f}   '
      f'sup Phi + log 2 = {phi + np.log(2):.3f}')
```

```
JS(p, q) = 0.193   JS(P, Q) = 0.319   sup Phi + log 2 = 0.319
```

The last two numbers agree. The value of the pairing game equals the Jensen–Shannon divergence between the two orderings P and Q, and the next subsection proves that this holds for every p and q.

16.3.2.3 The Lifted Game

The two-atom table treated the ordered pair as a single observation. In general, Φ is the expectation of a fixed function of the pair (x, x') under $p \otimes q$, which suggests doing the same. Let $P = p \otimes q$ and $Q = q \otimes p$ denote the two orderings of an independent real–fake pair, viewed as distributions on $\mathcal{X} \times \mathcal{X}$: under P the real sample sits first, under Q it sits second. These two distributions can play the log-loss game of Section 16.1 on the pair space, with a pair critic $\mathcal{D} : \mathcal{X} \times \mathcal{X} \to \mathbb{R}$ in place of D and the value function

$$V_{P,Q}(\mathcal{D}) = \mathbb{E}_{(a_1,a_2)\sim P}\big[\log \sigma(\mathcal{D}(a_1, a_2))\big] + \mathbb{E}_{(a_1,a_2)\sim Q}\big[\log \sigma(-\mathcal{D}(a_1, a_2))\big],$$

which is Equation 16.2 with (P, Q) substituted for (p, q). The following lemma connects that game to Φ.

Lemma (lifting). *Let $P = p \otimes q$ and $Q = q \otimes p$. Then: (i) the log ratio of the two orderings separates,* $\log \frac{dP}{dQ}(a_1, a_2) = \lambda(a_1) - \lambda(a_2)$*, an antisymmetric difference of single-sample scores; (ii) every difference critic $\mathcal{D}(a_1, a_2) = D(a_1) - D(a_2)$ satisfies $V_{P,Q}(\mathcal{D}) = 2\,\Phi(D)$; (iii) restricting the pair critic to differences does not lower the supremum,* $\sup_{\mathcal{D}} V_{P,Q}(\mathcal{D}) = \sup_{D} V_{P,Q}(D(a_1) - D(a_2))$.

Proof. *The product ratio separates.* The two product densities contain the same factors in opposite order:

$$\frac{dP}{dQ}(a_1, a_2) = \frac{p(a_1)\, q(a_2)}{q(a_1)\, p(a_2)} = \frac{\rho(a_1)}{\rho(a_2)}.$$

Taking logarithms proves (i).

The swap symmetry doubles the objective. Under P, the first term of $V_{P,Q}(\mathcal{D})$ is $\mathbb{E}[\log \sigma(D(a_1) - D(a_2))] = \Phi(D)$. Under Q, we have $a_1 \sim q$ and $a_2 \sim p$. Antisymmetry gives $-\mathcal{D}(a_1, a_2) = D(a_2) - D(a_1)$, and relabeling $(x, x') = (a_2, a_1)$ turns the second term into $\Phi(D)$ as well. Hence $V_{P,Q}(\mathcal{D}) = 2\Phi(D)$, proving (ii).

Differences suffice. Pointwise maximization of the log-loss game on $\mathcal{X} \times \mathcal{X}$ gives the optimal pair critic $\log(dP/dQ)$ wherever both densities are positive. Where one of them vanishes the optimal pair critic is infinite, and since $P > 0$ requires $p(a_1)\, q(a_2) > 0$, the difference $\lambda(a_1) - \lambda(a_2)$ takes the same infinite value there. By (i), this critic has the difference form with $D = \lambda$. Restricting the supremum to difference critics therefore does not change its value. ■

Part (iii) depends on the logistic payoff: the log ratio of two product measures is a sum of per-coordinate terms. Under other payoffs the best pair critic is a nonlinear function of that log ratio, and the relativistic objective only bounds its lifted divergence from below (Exercise 5). With the lemma in hand, the value of the pairing game follows from results already proved.

Theorem. *Define the pairing divergence* $d_{\mathrm{Rp}}(p, q) := \sup_D \Phi(D) + \log 2$. *Then*

$$d_{\mathrm{Rp}}(p, q) \;=\; \mathrm{JS}(p \otimes q,\; q \otimes p) \;=\; H\Big(\tfrac{1}{2}(p \otimes q + q \otimes p)\Big) - H(p) - H(q). \tag{16.19}$$

Proof. *Apply the value formula on the pair space.* Parts (ii) and (iii) of the lemma give $2 \sup_D \Phi(D) = \sup_{\mathcal{D}} V_{P,Q}(\mathcal{D})$. The value of the log-loss game, $\sup V = 2\,\mathrm{JS} - 2\log 2$ (Equation 16.4), applied to (P, Q) evaluates the right-hand side as $2\,\mathrm{JS}(P, Q) - 2\log 2$. Halving and adding $\log 2$ proves the first equality.

Collapse the pair entropies. The entropy form Equation 16.5, applied on $\mathcal{X} \times \mathcal{X}$, writes $\mathrm{JS}(P, Q)$ as the mixture entropy minus $\frac{1}{2}(H(P) + H(Q))$. Since each product measure has independent components, $H(P) = H(Q) = H(p) + H(q)$, which gives the second equality. This entropy calculation assumes the displayed entropies are finite; the measure-level first equality does not. ■

The additive constant in d_{Rp} calibrates the objective against a constant critic. Such a critic assigns probability one half to either ordering and obtains $\Phi = \log \sigma(0) = -\log 2$; it is optimal when $p = q$. Thus d_{Rp} measures the reduction in ordering uncertainty relative to random guessing. Jolicoeur-Martineau (2020) proved that this calibrated value is a divergence for a broad class of concave payoffs without computing it; her normalization is twice ours, so her divergence is $2\, d_{\mathrm{Rp}}$.

Substituting the optimal critic $D^\star = \lambda$ into Equation 16.17 directly, using $\sigma(\lambda(x) - \lambda(x')) = \rho(x)/(\rho(x) + \rho(x'))$, gives the same value in a form that is convenient for expansions:

$$d_{\mathrm{Rp}}(p, q) \;=\; \log 2 \;-\; \mathbb{E}_{x \sim p,\; x' \sim q}\Big[\log\Big(1 + \frac{q(x)\, p(x')}{p(x)\, q(x')}\Big)\Big]. \tag{16.20}$$

The mutual-information reading of Section 16.1 lifts along with the value. Draw $x \sim p$ and $x' \sim q$, present the two samples in a uniformly random order, and let the bit y record which position holds the real one, just as y labeled the origin of a single sample in Section 16.1. The ordered pair (a_1, a_2) is then a sample from the balanced mixture of P and Q with label

y, and the identification of the divergence with a mutual information, applied on the pair space, gives

$$d_{\mathrm{Rp}}(p,q) \;=\; I\big((a_1,a_2);\,y\big) \;=\; \log 2 - H(y \mid a_1,a_2). \tag{16.21}$$

The pairing divergence is the number of nats a randomly ordered pair carries about which member is real, just as $\mathrm{JS}(p,q)$ measures the information a single sample carries about its origin. The two-atom calculation above was this formula evaluated by hand.

16.3.2.4 Bounds, Local Behavior, and Saturation

The closed form permits an exact comparison between the pairing game and the corresponding single-sample game.

Proposition. *For all p and q:*

(a) $\mathrm{JS}(p,q) \;\le\; d_{\mathrm{Rp}}(p,q) \;\le\; \log 2$*;*

(b) if $q = p\,(1+\epsilon v)$ with $\mathbb{E}_p[v] = 0$ and v bounded, then $d_{\mathrm{Rp}}(p,q) = \frac{\epsilon^2}{4}\mathbb{E}_p[v^2] + O(\epsilon^3) = 2\,\mathrm{JS}(p,q) + O(\epsilon^3)$;

(c) if p and q have disjoint supports, then $d_{\mathrm{Rp}}(p,q) = \log 2$, independently of how far apart the supports lie.

Proof. (a) The map $(a_1,a_2) \mapsto a_1$ pushes P forward to p and Q forward to q. Since JS is an f-divergence, the data-processing inequality of Section 37.3.3.3 gives $\mathrm{JS}(p,q) \le \mathrm{JS}(P,Q) = d_{\mathrm{Rp}}$. Equation 16.21 gives the upper bound because $I\big((a_1,a_2);y\big) \le H(y) = \log 2$.

(b) In Equation 16.20, set $u = \lambda(x') - \lambda(x)$ and expand $\log(1+e^u) = \log 2 + \frac{u}{2} + \frac{u^2}{8} + O(u^4)$. The first moment is exact: $\mathbb{E}[u] = \mathbb{E}_q[\lambda] - \mathbb{E}_p[\lambda] = -D_{\mathrm{J}}$ with $D_{\mathrm{J}} = D_{\mathrm{KL}}(p\,\|\,q) + D_{\mathrm{KL}}(q\,\|\,p)$, and here $D_{\mathrm{J}} = \epsilon^2\mathbb{E}_p[v^2] + O(\epsilon^3)$. The second moment is $\mathbb{E}[u^2] = 2\epsilon^2\mathbb{E}_p[v^2] + O(\epsilon^3)$. Substitution gives $d_{\mathrm{Rp}} = \frac{\epsilon^2}{4}\mathbb{E}_p[v^2] + O(\epsilon^3)$. Applying the corresponding expansion to Equation 16.5 gives $\mathrm{JS} = \frac{\epsilon^2}{8}\mathbb{E}_p[v^2] + O(\epsilon^3)$. Exercise 2 supplies the moment calculations.

(c) A real-first pair lies in $\operatorname{supp} p \times \operatorname{supp} q$, whereas a fake-first pair lies in $\operatorname{supp} q \times \operatorname{supp} p$. These product sets are disjoint when the original supports are disjoint. Thus P and Q are mutually singular, the pair determines its ordering with certainty, and $\mathrm{JS}(P,Q) = \log 2$. ■

Part (a) says that a pair containing one sample from each distribution carries at least as much information about its ordering as a single sample carries about its origin. The gain is at most a factor of two: given the label, the two members are independent, so the pair carries $2\,\mathrm{JS}(p,q)$ minus the dependence between its members (Exercise 7). For $p \neq q$ the ratio $d_{\mathrm{Rp}}/\mathrm{JS}$ therefore lies in $[1,2]$ and, by part (b), tends to 2 near $q = p$; the two atoms gave 1.66. Part (c) shows that pairing does not resolve support saturation. For the point masses $p = \delta_0$ and $q_\theta = \delta_\theta$ of Section 16.1, the pairing value equals $\log 2$ for every $\theta \neq 0$ and gives the generator no gradient in θ.

16.3.2.5 Verification on Five Atoms

Because the theorem is an identity, a finite example can verify it without sampling error. On five atoms the critic is a vector $D \in \mathbb{R}^5$ and Φ is an exact sum over 25 pairs. We maximize Φ by gradient ascent along the functional derivative Equation 16.18 and compare the maximum plus $\log 2$ with both forms in Equation 16.19. The optimized critic should equal λ up to an additive constant.

```
p = np.array([0.35, 0.25, 0.20, 0.12, 0.08])   # data distribution
q = np.array([0.02, 0.05, 0.13, 0.30, 0.50])   # generator distribution
lam = np.log(p / q)

def sigma(t):
    return 1 / (1 + np.exp(-t))

D = np.zeros(5)
for _ in range(400):   # ascend Phi along its functional derivative
    R = sigma(D[None, :] - D[:, None])         # R[i, j] = sigma(D_j - D_i)
    D += 20.0 * (p * (R @ q) - q * (p @ R))
U = D[:, None] - D[None, :]                    # U[i, j] = D_i - D_j
phi = (np.outer(p, q) * (-np.logaddexp(0.0, -U))).sum()
P, Q = np.outer(p, q).ravel(), np.outer(q, p).ravel()
print(f'sup Phi + log 2, by gradient ascent:    {phi + np.log(2):.9f}')
print(f'JS(p (x) q, q (x) p), computed directly: {js(P, Q):.9f}')
print(f'entropy form H((P+Q)/2) - H(p) - H(q):  '
      f'{entropy((P + Q) / 2) - entropy(p) - entropy(q):.9f}')
print(f'JS(p, q) = {js(p, q):.6f},  log 2 = {np.log(2):.6f}')
D_c, lam_c = D - D.mean(), lam - lam.mean()
print(f'max |D - lambda|, both centered:        '
      f'{np.abs(D_c - lam_c).max():.1e}')
```

```
sup Phi + log 2, by gradient ascent:    0.376889913
JS(p (x) q, q (x) p), computed directly: 0.376889913
entropy form H((P+Q)/2) - H(p) - H(q):  0.376889913
JS(p, q) = 0.234050,  log 2 = 0.693147
max |D - lambda|, both centered:        4.4e-16
```

The three computations of d_{Rp} agree to nine decimal places, and after centering the recovered critic matches λ to double-precision round-off. The value respects part (a), $\mathrm{JS}(p, q) \approx 0.23 < d_{\mathrm{Rp}} \approx 0.38 < \log 2 \approx 0.69$, so here a pair carries about 1.6 times the information of a single sample.

16.3.3 Ranking and Mode Coverage

16.3.3.1 Convexity in the Generator's Distribution

Pairing made Φ bilinear in (p, q), so joint convexity is lost, but convexity in the generator's distribution, which the generator's minimization needs, survives. Grouping Equation 16.17 by the generated member gives

$$d_{\mathrm{Rp}}(p, q) \;=\; \log 2 + \sup_{D} \mathbb{E}_{x' \sim q}\big[T_D(x')\big], \qquad T_D(x') := \mathbb{E}_{x \sim p}\big[\log \sigma(D(x) - D(x'))\big],$$

a supremum of functionals linear in q, so $q \mapsto d_{\mathrm{Rp}}(p, q)$ is convex. Convexity in q is a statement about mixtures: along $q_t = (1-t)\, q + t\, p$ the divergence is at most $(1-t)\, d_{\mathrm{Rp}}(p, q)$. A generator moves samples instead, and a sample carried from one data point to another passes through positions that hold no data. Convexity in q therefore does not exclude local minima in the positions of the generated samples.

16.3.3.2 The Rank Weight

Differentiating Φ through $x' = G(z)$, as Section 16.1 did for the classical game, shows that minimizing Φ moves each generated sample up the critic's score surface, along $\nabla_\theta D(G(z))$, with a per-sample weight. The classical saturating loss moves it in the same direction with the weight $\sigma(D(x'))$. Side by side, the two weights are

$$w_{\mathrm{Rp}}(x') = \mathbb{E}_{x \sim p}\big[\sigma(D(x') - D(x))\big], \qquad w_{\mathrm{sat}}(x') = \sigma(D(x')). \tag{16.22}$$

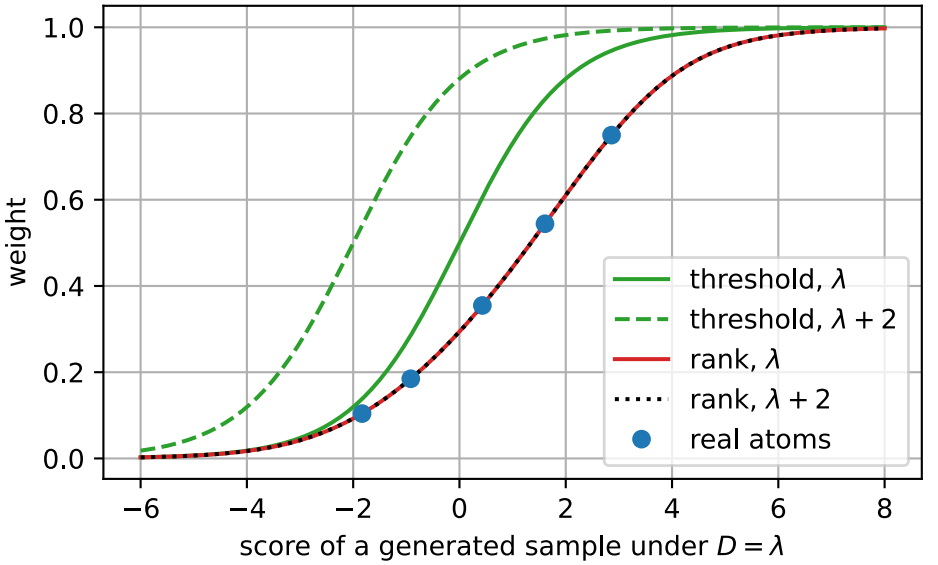
The classical weight compares a sample with the critic's zero level, which the pairing game leaves arbitrary. The relativistic weight, or *rank weight*, is the probability that x' outranks a random real sample. It is invariant to a shift of the critic and judges a generated sample against the whole real population.

On the five atoms of Section 16.3.2.5, take the critic $D = \lambda$ and the shifted critic $D = \lambda + 2$, which the pairing game cannot tell apart. A generated sample whose score under λ is t gets the threshold weight $\sigma(t)$ under the first critic and $\sigma(t+2)$ under the second; its rank weight $\sum_x p(x)\, \sigma(t - \lambda(x))$ is the same under both. The plot draws the four curves and marks the real atoms at their scores.

```
def rank_weight(t, D, p):   # E_{x ~ p}[sigma(t - D(x))], generated scores t
    return (p[:, None] * sigma(t[None, :] - D[:, None])).sum(axis=0)

ts = np.linspace(-6.0, 8.0, 281)   # scores of generated samples under lambda
d2l.plot([ts] * 4 + [lam],
         [sigma(ts), sigma(ts + 2), rank_weight(ts, lam, p),
          rank_weight(ts + 2, lam + 2, p), rank_weight(lam, lam, p)],
         r'score of a generated sample under $D = \lambda$', 'weight',
         legend=[r'threshold, $\lambda$', r'threshold, $\lambda + 2$',
                 r'rank, $\lambda$', r'rank, $\lambda + 2$', 'real atoms'],
         fmts=('C2-', 'C2--', 'C3-', 'k:', 'C0o'), figsize=(5.5, 3.2))
top = lam.max()   # the highest real score
print(f'highest real score: threshold {sigma(top):.2f} at lambda, '
      f'{sigma(top + 2):.2f} at lambda + 2; '
      f'rank {rank_weight(np.array([top]), lam, p)[0]:.2f} at both')
```

```
highest real score: threshold 0.95 at lambda, 0.99 at lambda + 2; rank 0.75 at both
```



Shifting the critic moves the threshold curve two units to the left, so at the highest real score the threshold weight rises from 0.95 to 0.99 although the pairing game has not changed. The two rank curves coincide. They rise smoothly across the real scores, each real atom contributing its mass $p(x)$, and reach only 0.75 at the highest one, so the rank weight separates samples that the threshold weight treats as nearly equal.

Neither weight by itself forbids collapse. Against a fixed critic, each generator loss is the q-average of a function that decreases as $D(x')$ grows, so it is linear in q and smallest when all generated mass sits at the critic's highest-scoring point. What differs is how the critic responds once the generator moves.

16.3.3.3 Paired Landscapes

Whether a collapsed generator is trapped depends on that response. Sun et al. (2020) settled the question in the smallest setting where it can be settled exactly: n distinct data points $x_1, \dots, x_n$, n generated points $x'_1, \dots, x'_n$ that the generator may place anywhere, and each loss evaluated at the critic's best response over all functions D:

$$\begin{aligned} \mathcal{E}_{\text{GAN}} &= \sup_D \tfrac{1}{2n} \textstyle\sum_i \left[\log \sigma(D(x_i)) + \log \sigma(-D(x'_i)) \right], \\ \mathcal{E}_{\text{diag}} &= \sup_D \tfrac{1}{n} \textstyle\sum_i \log \sigma(D(x_i) - D(x'_i)), \\ \mathcal{E}_{\text{all}} &= \sup_D \tfrac{1}{n^2} \textstyle\sum_{i,j} \log \sigma(D(x_i) - D(x'_j)). \end{aligned} \tag{16.23}$$

The first is the classical loss. The second, the *paired loss*, pairs the i-th real point with the i-th generated point, as a minibatch implementation does. The third averages over all n^2 cross pairs, so its generator weight is the rank weight. Each reaches its smallest value, $-\log 2$, when the generated points cover the data points one each. Sun et al. proved that every *collision*, a configuration with the generated points on data points and at least two of them coinciding, is a suboptimal strict local minimum of $\mathcal{E}_{\text{GAN}}$; there are $n^n - n!$ of them. For

$\mathcal{E}_{\text{diag}}$ they proved that from every configuration a path along which the loss never increases leads to a global minimum.

Two data points show the mechanism. Put both generated points on x_1. Half of the all-pairs comparisons are then ties, so $\mathcal{E}_{\text{all}} = -\frac{1}{2}\log 2$, and moving either point off x_1 raises the loss to $-\frac{1}{4}\log 2$. In the paired loss, x_2' sits on x_1 rather than on its partner x_2, so the critic scores it as fully fake and its term is already zero. It can travel to x_2 without raising the loss.

For larger n we compute. With $\hat{p}$ and $\hat{q}$ putting mass $1/n$ on each data point and each generated point, the value of the log-loss game gives $\mathcal{E}_{\text{GAN}} = \text{JS}(\hat{p}, \hat{q}) - \log 2$, and the theorem of this section gives $\mathcal{E}_{\text{all}} = \text{JS}(\hat{p} \otimes \hat{q}, \hat{q} \otimes \hat{p}) - \log 2$. For the paired loss, draw an arrow from each real point to the position of its generated partner. Arrows that can be ordered consistently are scored to zero. Arrows on a cycle cannot all point downhill, because the score differences around a cycle sum to zero, and the concavity of $\log \sigma$ makes ties best, at $-\log 2$ per pair. Hence $\mathcal{E}_{\text{diag}}$ is $-\log 2$ times the fraction of arrows on a cycle, counting a generated point on its own partner as a cycle of length one.

Real point i sits at position i and generated point i at position `gen[i]`. The function `losses` returns the three best-response losses of a configuration; we first check it against the two-point calculation above.

```
def on_cycle(i, gen):
    """Does the arrow from real point i to its partner's position close a cycle?"""
    j = gen[i]
    for _ in range(len(gen)):
        if j == i:
            return True
        if j not in range(len(gen)):  # a position that holds no data point
            return False
        j = gen[j]                    # follow the arrow of the real point at j
    return False

def losses(gen):
    """Best-response classical, all-pairs and paired losses of a configuration."""
    n = len(gen)
    support = sorted(set(range(n)) | set(gen))
    p_hat = np.isin(support, range(n)) / n
    q_hat = np.array([gen.count(s) for s in support]) / n
    P, Q = np.outer(p_hat, q_hat).ravel(), np.outer(q_hat, p_hat).ravel()
    paired = -np.log(2) * np.mean([on_cycle(i, gen) for i in range(n)])
    return np.array([js(p_hat, q_hat) - np.log(2), js(P, Q) - np.log(2), paired])

for gen in ([0, 0], [0, 0.1]):   # both on x_1; then x'_2 steps off it
    print(f'gen = {gen}: (classical, all pairs, paired) / log 2 = '
          f'{np.round(losses(gen) / np.log(2), 2)}')
```

```
gen = [0, 0]: (classical, all pairs, paired) / log 2 = [-0.69 -0.5  -0.5 ]
gen = [0, 0.1]: (classical, all pairs, paired) / log 2 = [-0.5  -0.25 -0.5 ]
```

Moving x_2' off x_1 raises the all-pairs loss from $-\frac{1}{2}\log 2$ to $-\frac{1}{4}\log 2$ and leaves the paired loss at $-\frac{1}{2}\log 2$, as computed by hand. A small move lets each generated point stay or step to a nearby position that holds no data, so coincident points may separate. The count keeps the assignments that every small move makes worse but that are not global minima.

```
def small_moves(gen):
    """Each generated point stays or steps to one of n fresh positions next to it."""
    n = len(gen)
    for steps in itertools.product(range(n + 1), repeat=n):
        if any(steps):
            yield [g + k / 10 if k else g for g, k in zip(gen, steps)]

def count_bad_minima(n):
    """Suboptimal strict local minima among the assignments, per loss."""
    best, count = losses(list(range(n))), np.zeros(3, dtype=int)
    for gen in itertools.product(range(n), repeat=n):
        value = losses(list(gen))
        rises = np.all([losses(m) > value + 1e-9 for m in small_moves(list(gen))],
                       axis=0)
        count += rises & (value > best + 1e-9)
    return count
```

At $n = 2$ the count should reproduce the hand calculation above; at $n = 3$ it checks the theorem.

```
for n in (2, 3):
    c = count_bad_minima(n)
    print(f'n = {n}: classical {c[0]}, all pairs {c[1]}, paired {c[2]};  '
          f'n^n - n! = {n**n - math.factorial(n)}')
```

```
n = 2: classical 2, all pairs 2, paired 0;  n^n - n! = 2
n = 3: classical 21, all pairs 21, paired 0;  n^n - n! = 21
```

At $n = 2$ and 3 the classical and all-pairs losses each have the $n^n - n!$ collision minima, and the paired loss has none. The guarantee belongs to the one-to-one coupling, not to the rank weight: the all-pairs loss ranks each generated sample against all real ones and still has 21 minima at $n = 3$, although it is convex in $\hat{q}$ along mixtures.

Minibatch training, however, redraws the pairs at every step, and averaged over all $n!$ pairings the paired objective is exactly the all-pairs objective. A critic that responds to that average faces as many collapse minima as the classical loss, so the one-to-one guarantee belongs to a fixed coupling (Exercise 8).

16.3.3.4 Saturating and Non-Saturating Generators

The rank weight still saturates. If $D(x')$ is far below every real score, then $w_{\mathrm{Rp}}(x') \approx 0$, so the generated samples that need the largest change receive the smallest updates. As in the classical game, maximizing $\mathbb{E}[\log \sigma(D(x') - D(x))]$ instead of minimizing Φ keeps the update direction and gives it the complementary weight:

generator objective	weight on the update of x'	on a badly ranked sample
zero-sum: minimize Φ	$\mathbb{E}_{x\sim p}[\sigma(D(x') - D(x))]$	weight $\to 0$: the update vanishes
non-saturating: maximize $\mathbb{E}[\log \sigma(D(x') - D(x))]$	$\mathbb{E}_{x\sim p}[\sigma(D(x) - D(x'))]$	weight $\to 1$: the update is largest

Jolicoeur-Martineau (2019) trains the generator on the non-saturating form. The R3GAN recipe of Huang et al. (2024), the modern baseline of the next two sections, writes its objective as the zero-sum minimax over Φ, and implementing that equation literally puts the generator on the saturating row. Train the generator on the non-saturating form instead, as the R3GAN reference implementation does.

The two variants share their fixed point. At $D^\star = \lambda$ the function $u \mapsto \log(1+e^u)$ is convex, and Jensen's inequality bounds the non-saturating pairing loss from below:

$$-\mathbb{E}_{x\sim p,\, x'\sim q}\big[\log\sigma(\lambda(x')-\lambda(x))\big] = \mathbb{E}\big[\log\big(1+e^{\lambda(x)-\lambda(x')}\big)\big] \;\geq\; \log\big(1+e^{D_{\mathrm{J}}}\big),$$

where $D_{\mathrm{J}} = \mathbb{E}_p[\lambda] - \mathbb{E}_q[\lambda] = D_{\mathrm{KL}}(p\,\|\,q) + D_{\mathrm{KL}}(q\,\|\,p) \geq 0$. The bound exceeds $\log 2$ whenever $q \neq p$, and the loss equals $\log 2$ at $q = p$, so the non-saturating generator, too, is minimized only at the data. The variants differ when generated samples rank badly: at the optimal critic, the q-average of w_{Rp} is at most $\frac{1}{2}$ and tends to zero as the supports separate (Exercise 4). On the two atoms it is about 0.24, against 0.76 for the non-saturating weight.

16.3.3.5 Diagonal and All-Pairs Estimates

From n real and n generated samples, Φ can be estimated on the n aligned pairs or on all n^2 cross pairs. The all-pairs average has the smaller variance, yet in the experiments of Jolicoeur-Martineau (2020) it did not improve the generator. The cheaper diagonal estimate is the standard choice, and it is the paired loss the rest of the chapter trains with.

We can now compute the value of a pairing game and read a critic through both weights. Pairing keeps the optimal critic and the fixed point, lets a pair carry up to twice the information of a sample, and turns the generator's weight into a rank. It leaves the value at $\log 2$ on separated supports, and it does not make gradient training converge; Section 16.4 takes up both.

16.3.4 Summary

The pairing objective is the Bradley–Terry log-likelihood of ranking a real sample above a generated one. Its optimal critic is still the log density ratio, now up to a constant. Treating the ordered pair as one observation turns it into the log-loss game between $p \otimes q$ and $q \otimes p$, so its value is $\mathrm{JS}(p\otimes q, q\otimes p)$, the information a randomly ordered pair carries about which member is real: between $\mathrm{JS}(p,q)$ and $\log 2$, twice JS near $q = p$, and $\log 2$ on disjoint supports. The generator's weight becomes the rank of a generated sample among the real scores. With best-response critics, a fixed one-to-one coupling removes the collapse minima that the classical and all-pairs losses have, but pairs redrawn at every step average back to the all-pairs loss. The generator should be trained on the non-saturating form.

16.3.5 Exercises

1. **Shift invariance in training.** Adding a constant to the critic leaves Φ unchanged (Figure 16.5), so nothing in the pairing game fixes the level of the critic's scores.
    1. Show that the functional derivative Equation 16.18 integrates to zero over t for every critic D. What does this imply for `D.mean()` during the ascent loop of Section 16.3.2.5? Print it to check.

   2. Let a network critic end in a scalar output bias, $D(x) = D_0(x) + b$. Show that every minibatch estimate of Φ, diagonal or all-pairs, has zero gradient with respect to b. What then determines the level of the critic's scores during training?
   3. For a classical critic, the fraction of generated samples with $D(x') > 0$ is the fraction that the critic classifies as real. Why is this statistic uninformative for a critic trained on Φ, and which statistic of the same minibatch scores replaces it?
2. **Local expansion.** Let $q = p\,(1 + \epsilon v)$ with $\mathbb{E}_p[v] = 0$ and v bounded.
   1. Starting from Equation 16.20, write $u = \lambda(x') - \lambda(x)$ and expand $\log(1 + e^u) = \log 2 + \frac{u}{2} + \frac{u^2}{8} + O(u^4)$. Compute $\mathbb{E}[u] = \mathbb{E}_q[\lambda] - \mathbb{E}_p[\lambda]$ and $\mathbb{E}[u^2]$ to second order in ϵ, and conclude that $d_{\mathrm{Rp}} = \frac{\epsilon^2}{4}\mathbb{E}_p[v^2] + O(\epsilon^3)$.
   2. Apply the corresponding expansion to the entropy form Equation 16.5 to show that $\mathrm{JS}(p, q) = \frac{\epsilon^2}{8}\mathbb{E}_p[v^2] + O(\epsilon^3)$, so that locally $d_{\mathrm{Rp}} = 2\,\mathrm{JS} + O(\epsilon^3)$.
3. **Two Gaussians.** Let $p = \mathcal{N}(0, I)$ and $q = \mathcal{N}(\mu, I)$ on $\mathbb{R}^d$.
   1. Show that λ is affine in x and that $u = \lambda(x') - \lambda(x)$ in Equation 16.20 is Gaussian. Find its mean and variance.
   2. Write $\mathrm{JS}(p, q)$ in the same form, as $\log 2$ minus the expectation of $\log(1 + e^t)$ for a Gaussian t. Find the constant c for which $\mathrm{JS}(p, \mathcal{N}(c\mu, I)) = d_{\mathrm{Rp}}(p, q)$ holds for every μ.
   3. Evaluate d_{Rp} and JS by one-dimensional quadrature for $\|\mu\| \in \{0.1, 1, 2, 4\}$. Relate the ratio $d_{\mathrm{Rp}}/\mathrm{JS}$ to the local limit and to the saturation of Section 16.3.2.4.
4. code **Zero-sum and non-saturating weights.** The zero-sum generator updates each generated sample with the weight w_{Rp} of Equation 16.22, the non-saturating generator with $1 - w_{\mathrm{Rp}}(x')$.
   1. Treat each of the five atoms of Section 16.3.2.5 as a generated sample and evaluate both weights with the critic `D_c`. On which atoms does each variant place its largest weight, and how do these relate to the atoms where q exceeds p? By what factor does the q-average of the non-saturating weight exceed that of w_{Rp}?
   2. Let $D = \lambda$ and write $P = p \otimes q$ and $Q = q \otimes p$. Show that the q-average of w_{Rp} equals $\iint \frac{PQ}{P+Q}\,da_1\,da_2$, and that it is at most $\frac{1}{2}$, with equality only at $q = p$. What does it tend to as the supports separate, and what does this imply for the zero-sum generator early in training?
5. **Separability of the pair critic.** ▨ Replace the logistic payoff in Equation 16.17 by a concave payoff ℓ, giving $\Phi_\ell(D) = \mathbb{E}_{x\sim p,\,x'\sim q}[\ell(D(x) - D(x'))]$. On the pair space an unrestricted critic $\mathcal{D}$ collects $\ell(\mathcal{D})$ on pairs from P and $\ell(-\mathcal{D})$ on pairs from Q. By part (ii) of the lifting lemma, which holds for every ℓ, difference critics earn $2\,\Phi_\ell(D)$; by part (iii) they lose nothing for the logistic payoff.
   1. For $\ell(t) = -(1-t)^2$, show that the pointwise optimum of the pair-space game is $2\eta - 1 = \tanh\left(\frac{\lambda(a_1) - \lambda(a_2)}{2}\right)$ with $\eta = P/(P + Q)$. Show that it is not additively separable, and conclude that a relativistic least-squares game over difference critics optimizes a strict lower bound on its lifted divergence in general.
   2. Measure this gap on the five atoms of Section 16.3.2.5: evaluate the unrestricted value at the pointwise optimum, and maximize $2\,\Phi_\ell(D)$ over $D \in \mathbb{R}^5$ by gradient ascent. How large is the gap relative to the unrestricted value?
   3. Return to the logistic payoff and present the pair real-first with probability $\alpha \neq \frac{1}{2}$, so that the log loss weights pairs from P by α and pairs from Q by $1 - \alpha$. Derive the Bayes-optimal pair critic. Can a difference critic attain it? Which extension of the difference class can?
6. **More negatives.** ▨ Draw K real samples $x_1, \ldots, x_K \sim p$ and one generated sample

$x' \sim q$, place all $K+1$ samples in a uniformly random order $(a_1, \dots, a_{K+1})$, and let the critic predict which position holds the generated sample.

1. Show that the Bayes posterior for position j is the softmax $e^{-\lambda(a_j)} / \sum_k e^{-\lambda(a_k)}$, so that a single-sample score again suffices.
2. Repeat the lifting argument with $K+1$ hypotheses $P_j = p^{\otimes(j-1)} \otimes q \otimes p^{\otimes(K+1-j)}$. Using $H(P_j) = K\,H(p) + H(q)$, show that the chance-level log loss $\log(K+1)$ minus the optimal expected log loss is

$$d_K(p,q) = H\Big(\tfrac{1}{K+1} \textstyle\sum_j P_j\Big) - K\,H(p) - H(q), \tag{16.24}$$

and that $0 \le d_K \le \log(K+1)$.

3. Compare the game with the InfoNCE loss Equation 37.77 of Section 37.7.4. Which sample and which densities play the roles of the positive pair, the negatives, and the density ratio that the optimal InfoNCE critic recovers?

7. **Gap to twice Jensen–Shannon.** ▯ Take the randomly ordered pair (a_1, a_2) and its label y from Equation 16.21.
   1. Show that a_1 and a_2 are conditionally independent given y, and that each position alone carries $I(a_k; y) = \mathrm{JS}(p,q)$.
   2. Use the chain rule of mutual information to show that $d_{\mathrm{Rp}}(p,q) = 2\,\mathrm{JS}(p,q) - I(a_1; a_2)$.
   3. Apply the data-processing inequality to the Markov chain $a_1 - y - a_2$ to obtain $\mathrm{JS}(p,q) \le d_{\mathrm{Rp}}(p,q)$ without the pushforward argument. What is $I(a_1; a_2)$ when p and q have disjoint supports?
   4. Compute $I(a_1; a_2)$ for the five atoms of Section 16.3.2.5 from the 25-atom distribution $\frac{1}{2}(P+Q)$, and check that $2\,\mathrm{JS}(p,q) - I(a_1; a_2)$ reproduces the printed value of d_{Rp}.
8. code **Random re-pairing.** ▯▯ The enumeration of Section 16.3.3.3 fixes the coupling: the i-th real point is always paired with the i-th generated point. Minibatch training draws a new pairing at every step.
   1. Extend the count to $n = 4$ (256 assignments). Do the classical and all-pairs counts still equal $n^n - n!$, and does the paired loss still have none?
   2. Model a critic that best-responds to each pairing separately: for a permutation π, let $\mathcal{E}_\pi = \sup_D \frac{1}{n} \sum_i \log \sigma(D(x_i) - D(x'_{\pi(i)}))$, and let the generator minimize the average of $\mathcal{E}_\pi$ over all $n!$ permutations. Adapt `losses` and `count_bad_minima` to count the suboptimal strict local minima of this average for $n = 3$. For $n = 2$, trace what happens at the collision $x'_1 = x'_2 = x_1$ when one generated point moves away.
   3. A critic trained over many minibatches responds instead to the pairing-averaged objective, whose best response is $\mathcal{E}_{\mathrm{all}}$. Which model is closer to minibatch training with a fresh pairing per step, and what do your counts suggest about the landscape guarantee in training?

16.4 Gradient Penalties and Convergence

The garment generator of Section 16.1 learned to draw shirts, shoes and dresses, but its losses never settled. The critic's loss stayed well below $\log 2$ and the generator's well above it, both still drifting when training stopped. At the equilibrium, where the generated images match the data and the critic can only guess, both losses equal $\log 2$. The last three sections computed what the game is worth when the critic answers optimally. Training never gives

the critic that chance: both networks take small gradient steps at the same time, and each step of one player changes the objective of the other. Knowing where the equilibrium lies does not tell us whether these steps lead there.

Mescheder et al. (2018) asked which training methods for GANs converge. Their answer starts from the smallest game there is: one data point, one generated point and a straight-line critic. Plain gradient steps fail even on this game, and a penalty on the critic's slope at the data and generated samples repairs them. We analyze that game, add the penalty, ask what the penalized game measures, and train the resulting R3GAN loss (Huang et al. 2024) on a grid of 25 Gaussians.

16.4.1 The Dirac-GAN

The smallest game can be solved exactly, and it shows both obstacles of this section: its value is flat in the generator's position, and gradient steps rotate around the solution instead of approaching it.

16.4.1.1 The Game on Two Points

Place the data at the origin and let the generator produce a single point on the real line: $p = \delta_0$ and $q_\theta = \delta_\theta$, so $q = p$ exactly when $\theta = 0$. Let the critic be linear, $D_\psi(x) = \psi x$. Section 16.2 scored a real sample x by $\ell(D(x))$ and a generated sample x' by $\ell(-D(x'))$ for a classification loss ℓ, giving the margin objective

$$V_\ell(D) = \mathbb{E}_{x\sim p}\big[\ell(D(x))\big] + \mathbb{E}_{x'\sim q}\big[\ell(-D(x'))\big].$$

For two point masses both expectations are single evaluations:

$$V(\theta,\psi) \;=\; \ell\big(D_\psi(0)\big) + \ell\big(-D_\psi(\theta)\big) \;=\; \ell(0) + \ell(-\psi\theta). \tag{16.25}$$

This is the *Dirac-GAN* of Mescheder et al. (2018) . Any differentiable loss with $\ell'(0) > 0$ works. We use the logistic loss $\ell = \log\sigma$, whose derivatives are $\ell'(t) = \sigma(-t)$ and $\ell''(t) = -\sigma(t)\sigma(-t)$, so $\ell'(0) = \frac{1}{2}$ and $\ell''(0) = -\frac{1}{4}$.

For every $\theta \neq 0$ the two point masses have disjoint supports. The critic's best response sends $\psi\theta \to -\infty$, so $\sup_\psi V = \ell(0) + \sup_t \ell(t)$, the same number for every $\theta \neq 0$. A value that does not depend on θ gives the generator no gradient, however close to the data it starts. This is the saturation of Section 16.1 in its smallest form.

16.4.1.2 Circles in Continuous Time

Training does not compute a best response. Both players take gradient steps from where they are. The game has one stationary point, $(\theta,\psi) = (0,0)$, where the generated point matches the data and the critic is flat. In the simultaneous gradient flow the generator descends V and the critic ascends it:

$$\dot\theta \;=\; -\partial_\theta V \;=\; \psi\,\ell'(-\psi\theta), \qquad \dot\psi \;=\; +\partial_\psi V \;=\; -\theta\,\ell'(-\psi\theta). \tag{16.26}$$

Both components are the same positive scalar $\ell'(-\psi\theta)$ times the vector $(\psi, -\theta)$, which is orthogonal to the position (θ, ψ). The squared distance to the equilibrium therefore does not change:

$$\frac{d}{dt}(\theta^2 + \psi^2) = 2\theta\psi\,\ell'(-\psi\theta) - 2\psi\theta\,\ell'(-\psi\theta) = 0.$$

Every trajectory is a circle around the solution, for every loss in this family. The linearization at the origin gives the same picture locally,

$$J = \begin{pmatrix} 0 & \ell'(0) \\ -\ell'(0) & 0 \end{pmatrix}, \qquad \nu_{1,2} = \pm\, i\,\ell'(0), \tag{16.27}$$

with purely imaginary eigenvalues: the equilibrium is a center, not an attractor. The generated point passes the data again and again without settling there. The common factor $\ell'(-\psi\theta) = \sigma(D_\psi(\theta))$ is the weight with which the saturating generator of Section 16.1 scales the update of a generated sample. Because it multiplies both components, it sets how fast the parameters turn and not where they go, so no reweighting of this kind turns the circle into a spiral toward the solution.

16.4.1.3 Outward Spirals in Discrete Time

Discrete steps do worse than the flow. Write $\mathbf{u}(\theta, \psi)$ for the right-hand side of Equation 16.26. A simultaneous step of size $h > 0$ maps (θ, ψ) to $(\theta, \psi) + h\,\mathbf{u}$. Since $\mathbf{u}$ is orthogonal to (θ, ψ), the Pythagorean theorem gives

$$\big\|(\theta, \psi) + h\,\mathbf{u}\big\|^2 \;=\; \big\|(\theta, \psi)\big\|^2 + h^2\big\|\mathbf{u}\big\|^2,$$

so each step adds $h^2\|\mathbf{u}\|^2$ to the squared distance. For the logistic loss $\mathbf{u}$ vanishes only at the equilibrium, so every step from any other point moves away from the solution, and a smaller step only slows the outward spiral (exercise 1). Updating the two players in turn instead of simultaneously does not restore convergence either (exercise 5).

The two repairs of the earlier sections leave these dynamics in place. The non-saturating generator has the same Jacobian J at the equilibrium; away from it the field gains a slow inward drift, which discrete steps stop on a cycle of radius about $\sqrt{2h}$ (exercise 9). The pairing objective of Section 16.3 scores the pair by $\ell(D_\psi(0) - D_\psi(\theta)) = \ell(-\psi\theta)$, which differs from Equation 16.25 only by the constant $\ell(0)$, so its gradient field, its circles and its spirals are the classical ones. A cure has to change the dynamics themselves.

16.4.2 Zero-Centered Penalties

The equilibrium attracts the iterates only if the eigenvalues of the linearized dynamics have negative real parts. The rotation needs friction, and the natural place to add it is the critic's slope, the quantity ψ that keeps the rotation going. Define the two *zero-centered gradient penalties*

$$R_1 \;=\; \frac{\gamma}{2}\,\mathbb{E}_{x\sim p}\Big[\big\|\nabla_x D(x)\big\|^2\Big], \qquad R_2 \;=\; \frac{\gamma}{2}\,\mathbb{E}_{x'\sim q}\Big[\big\|\nabla_{x'} D(x')\big\|^2\Big], \tag{16.28}$$

with weight $\gamma > 0$, and let the critic maximize $V_\ell(D) - R_1 - R_2$ while the generator's objective stays unchanged (Roth et al. 2017; Mescheder et al. 2018). The penalties act on the critic's gradient with respect to its *input*, not its parameters: R_1 limits how steeply the critic varies at data samples and R_2 at generated samples. They are called zero-centered because the flat critic $\nabla_x D = 0$ minimizes both. At $q = p$ the optimal critic $\log(p/q)$ of Section 16.1 is the constant zero, so the penalties vanish at the equilibrium they are meant to stabilize. Since $\frac{1}{2}(\mathbb{E}_p + \mathbb{E}_q) = \mathbb{E}_m$, their sum is

$$R_1 + R_2 \;=\; \gamma\, \mathbb{E}_{x\sim m}\Big[\big\|\nabla_x D(x)\big\|^2\Big], \qquad m = \frac{p+q}{2}, \tag{16.29}$$

one smoothness penalty under the mixture m, which constrains the critic wherever either distribution puts mass.

16.4.2.1 Damping the Dirac-GAN

For the linear critic, $\nabla_x D_\psi = \psi$ at every x, so both penalties equal $\frac{\gamma}{2}\psi^2$. Subtracting one of them from the critic's objective adds $-\gamma\psi$ to the critic's update:

$$\dot\theta = \psi\,\ell'(-\psi\theta), \qquad \dot\psi = -\theta\,\ell'(-\psi\theta) - \gamma\psi.$$

Using both penalties replaces γ by 2γ. Only the lower-right entry of the Jacobian changes:

$$J_\gamma = \begin{pmatrix} 0 & \ell'(0) \\ -\ell'(0) & -\gamma \end{pmatrix}, \qquad \nu_{1,2} \;=\; -\frac{\gamma}{2} \pm \sqrt{\frac{\gamma^2}{4} - \ell'(0)^2}. \tag{16.30}$$

Both eigenvalues have negative real parts for every $\gamma > 0$. The equilibrium attracts, and small enough gradient steps converge to it at a linear rate (Mescheder et al. 2018). As for a damped spring, there are three regimes. For $\gamma < 2\ell'(0)$ the square root is imaginary and trajectories spiral inward. At $\gamma = 2\ell'(0)$, which is $\gamma = 1$ for the logistic loss, the repeated real eigenvalue gives critical damping, the fastest approach without oscillation. Larger values overdamp the dynamics: the slower eigenvalue, close to $-\ell'(0)^2/\gamma$, makes convergence slower as γ grows.

16.4.2.2 Phase Portraits

We check the circles, the outward spiral and the damping by simulating the two equations for the logistic loss. The function `trajectory` repeats one update of step size h: `rk4_step` integrates the flow with a fourth-order Runge–Kutta step, and `euler_step` takes one simultaneous gradient step.

```
def dirac_field(theta, psi, gamma=0.0):
    """Simultaneous gradient field of the Dirac-GAN with the logistic loss."""
    w = 1.0 / (1.0 + np.exp(-psi * theta))      # l'(-psi theta) = w_sat
    return np.array([psi * w, -theta * w - gamma * psi])

def euler_step(x, h, gamma):                    # one simultaneous step
    return x + h * dirac_field(*x, gamma)
```

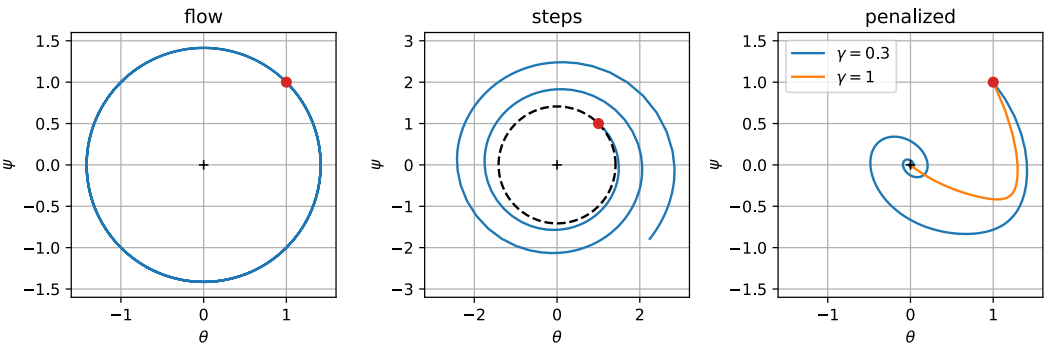
```
def rk4_step(x, h, gamma):                          # one RK4 step of the ODE
    k1 = dirac_field(*x, gamma)
    k2 = dirac_field(*(x + h / 2 * k1), gamma)
    k3 = dirac_field(*(x + h / 2 * k2), gamma)
    k4 = dirac_field(*(x + h * k3), gamma)
    return x + h / 6 * (k1 + 2 * k2 + 2 * k3 + k4)

def trajectory(step, x0, h, steps, gamma=0.0):
    xs = [np.array(x0)]
    for _ in range(steps):
        xs.append(step(xs[-1], h, gamma))
    return np.stack(xs)
```

All panels start at $(\theta, \psi) = (1, 1)$, in red; the cross marks the equilibrium. They show the flow, 300 simultaneous steps of size $h = 0.2$ against the flow's circle (dashed), and the same steps with the penalty at $\gamma = 0.3$, in the underdamped regime, and at the critical $\gamma = 1$.

```
def phase_panel(ax, paths, title, lim, legend=None, fmts=('C0-', 'C1-')):
    """Draw paths in the (theta, psi) plane, with start and equilibrium."""
    X = [P[:, 0] for P in paths] + [[paths[0][0, 0]], [0.0]]
    Y = [P[:, 1] for P in paths] + [[paths[0][0, 1]], [0.0]]
    d2l.plot(X, Y, r'$\theta$', r'$\psi$', legend, xlim=(-lim, lim),
             ylim=(-lim, lim), fmts=fmts[:len(paths)] + ('C3o', 'k+'), axes=ax)
    ax.set_title(title)
    ax.set_aspect('equal')

x0, t = (1.0, 1.0), np.linspace(0, 2 * np.pi, 200)
circle = np.sqrt(2) * np.stack([np.cos(t), np.sin(t)], axis=1)
fig, axes = d2l.plt.subplots(1, 3, figsize=(9.5, 3.4))
phase_panel(axes[0], [trajectory(rk4_step, x0, 0.05, 800)], 'flow', 1.6)
phase_panel(axes[1], [trajectory(euler_step, x0, 0.2, 300), circle], 'steps',
            3.2, fmts=('C0-', 'k--'))
phase_panel(axes[2], [trajectory(euler_step, x0, 0.2, 300, gamma=0.3),
                      trajectory(euler_step, x0, 0.2, 300, gamma=1.0)],
            'penalized', 1.6, legend=[r'$\gamma = 0.3$', r'$\gamma = 1$'])
fig.tight_layout()
```



The integrated flow retraces one circle, as the conservation law requires. The discrete iterates leave the dashed circle at once and move outward on every revolution; after 300 steps

their distance from the equilibrium has doubled. Both penalized trajectories converge, the underdamped one along a shrinking spiral and the critically damped one directly. The penalty turns divergence into convergence and leaves the equilibrium where it was.

16.4.2.3 One-Centered versus Zero-Centered

WGAN-GP centers its penalty differently (Gulrajani et al. 2017). To enforce the Lipschitz constraint of Section 16.2, it penalizes $\mathbb{E}\big[(\|\nabla_x D\| - 1)^2\big]$ at points between real and generated samples, because optimal W_1 critics have unit slope along transport rays. At $q = p$, however, the optimal critic is constant. A penalty that prefers unit slope keeps pushing a generator that already matches the data, and on the Dirac-GAN these dynamics do not converge (Mescheder et al. 2018) (exercise 7).

penalty	critic it prefers at $q = p$	Dirac-GAN
one-centered, $\mathbb{E}\big[(\|\nabla_x D\| - 1)^2\big]$	unit slope	does not converge
zero-centered, $\mathbb{E}_m\big[\|\nabla_x D\|^2\big]$	flat	converges for every $\gamma > 0$

16.4.2.4 One Penalty or Two

The Dirac-GAN cannot tell R_1 from R_2, because a linear critic has the same slope everywhere. Near equilibrium the supports of p and q nearly coincide, so either penalty suffices, and the local convergence result of Mescheder et al. (2018) needs only one. Far from equilibrium the two differ. Roth et al. (2017) showed that penalizing $\mathbb{E}_p\big[\|\nabla_x D\|^2\big]$ changes the game, to leading order, as if p were convolved with Gaussian noise of covariance proportional to γ, and R_2 smooths q in the same way. Smoothed, two distributions with disjoint supports overlap, and the divergence responds to the distance between them. With R_1 alone the critic is unconstrained where only generated samples lie. R3GAN's ablation on StackedMNIST, stacked MNIST digits with 1,000 known modes (Metz et al. 2017), shows the consequence (Huang et al. 2024):

loss	modes recovered (of 1,000)	reverse KL
RpGAN with R_1 and R_2	1,000	0.078
GAN with R_1 and R_2	693	0.927
RpGAN with R_1 only	diverged	—
GAN with R_1 only	diverged	—

With R_1 alone the critic's input gradients on generated samples grew and both losses diverged, for every γ from 0.1 to 100 in the paired case. With both penalties both trained, and the paired loss recovered every mode: in this ablation the penalties decided convergence and the pairing decided coverage.

16.4.3 The Penalized Value Near Equilibrium

The eigenvalues say how training approaches the equilibrium; what does the penalized game measure on the way? Without the penalty the value was a divergence of the density

ratio, blind to how far apart separated supports are. With it, the value near equilibrium blends that divergence with a transport distance, and γ decides which length scales fall on which side.

16.4.3.1 The Local Value

Near $q = p$ the optimal critic is close to zero, so we expand the payoff to second order in the critic. With $\langle p - q, D\rangle = \mathbb{E}_p[D] - \mathbb{E}_q[D]$, and up to third-order terms,

$$V_\ell(D) = 2\ell(0) + \ell'(0)\,\langle p - q,\, D\rangle - |\ell''(0)| \int m\, D^2.$$

The pairing objective has $\mathrm{Var}_m[D]$ in place of $\int mD^2$, which gives the same supremum once the critic's constant is chosen best. The quadratic term is not negligible: the best critic has size proportional to $p - q$, so all terms are of the same order at the optimum.

Proposition. *Let $\mathcal{L}_m D = -\nabla \cdot (m \nabla D)$. Near equilibrium, the penalized critic problem has the value*

$$\sup_D \Big\{ \ell'(0)\,\langle p - q,\, D\rangle - |\ell''(0)| \int m\, D^2 - \gamma \int m\, \|\nabla_x D\|^2 \Big\} \;=\; \frac{\ell'(0)^2}{4}\,\big\langle p - q,\, (|\ell''(0)|\, m + \gamma \mathcal{L}_m)^{-1}(p - q)\big\rangle \tag{16.31}$$

attained by the critic that solves $(|\ell''(0)|\, m + \gamma \mathcal{L}_m) D = \frac{\ell'(0)}{2}(p - q)$.

Proof. Write $\mathcal{A} = |\ell''(0)|\, m + \gamma \mathcal{L}_m$; integrating by parts, the two quadratic terms are $\langle D, \mathcal{A} D\rangle$, and $\mathcal{A}$ is symmetric and positive. Along a ray $\{t D_0\}$ the objective $t\,\ell'(0)\langle p - q, D_0\rangle - t^2 \langle D_0, \mathcal{A} D_0\rangle$ is a concave parabola in t with maximum $\ell'(0)^2 \langle p - q, D_0\rangle^2 / (4 \langle D_0, \mathcal{A} D_0\rangle)$. By the Cauchy–Schwarz inequality in the inner product $\langle \cdot, \mathcal{A}\, \cdot \rangle$, the largest value over directions is $\frac{\ell'(0)^2}{4} \langle p - q, \mathcal{A}^{-1}(p - q)\rangle$, reached at $D_0 \propto \mathcal{A}^{-1}(p - q)$. ∎

The value is a weighted squared size of $p - q$. As $\gamma \to 0$ it becomes $\frac{\ell'(0)^2}{4|\ell''(0)|} \int (p - q)^2 / m$, which for the logistic loss is $\frac{1}{4} \int (p - q)^2 / m$, the local form of $2\,\mathrm{JS}(p, q)$, the unpenalized value of Section 16.1. When γ is large, the penalty dominates and the value approaches

$$\frac{\ell'(0)^2}{4\gamma} \|p - q\|^2_{\dot{H}^{-1}(m)}, \qquad \|v\|_{\dot{H}^{-1}(m)} = \sup \Big\{ \langle v, D\rangle : \int m \|\nabla_x D\|^2 \le 1 \Big\}, \tag{16.32}$$

a squared dual Sobolev norm, taken over critics modulo constants.

16.4.3.2 A Gaussian Shift

For $p = \mathcal{N}(0, 1)$ and $q = \mathcal{N}(\mu, 1)$ with a small shift μ, to first order $p - q \approx -\mu\, x\, m(x)$. Since $\mathcal{L}_m x = x\, m(x)$ for the standard normal density m, a straight-line critic solves the equation of the proposition:

$$D(x) = -\frac{\ell'(0)\,\mu}{2\,(|\ell''(0)| + \gamma)}\, x, \qquad \text{value} = \frac{\ell'(0)^2 \mu^2}{4\,(|\ell''(0)| + \gamma)} = \frac{\mu^2}{4 + 16\gamma} \text{ for the logistic loss.}$$

This is the log ratio $-\mu x$ with its slope shrunk by $1 + 4\gamma$. At $\gamma = 0$ the value is $\mu^2/4 = 2$ JS to leading order, and for large γ it is $\mu^2/(16\gamma) = \frac{\ell'(0)^2}{4\gamma} W_2(p, q)^2$, since the Wasserstein-2 distance of a shift is $|\mu|$. To check both approximations, we solve the full penalized problem by Newton's method for a critic tabulated on a grid.

```
def penalized_value(p, q, gamma, x, newton_steps=30):
    """Value of the logistic game minus gamma E_m[D'^2], on a grid."""
    dx, m = x[1] - x[0], (p + q) / 2
    diff = np.diff(np.eye(len(x)), axis=0) / dx       # D -> D'
    K = 2 * gamma * dx * diff.T @ (((m[1:] + m[:-1]) / 2)[:, None] * diff)
    D = np.zeros_like(x)
    for _ in range(newton_steps):                     # Newton ascent on D
        s = 1 / (1 + np.exp(-D))                      # sigma(D)
        grad = (p * (1 - s) - q * s) * dx - K @ D     # gradient of the value
        neg_hess = np.diag(2 * m * s * (1 - s) * dx) + K  # minus its Hessian
        D = D + np.linalg.solve(neg_hess, grad)
    value = -(p * np.logaddexp(0, -D) + q * np.logaddexp(0, D)).sum() * dx
    return value - D @ K @ D / 2 + 2 * np.log(2)       # minus 2 l(0)

x, mu = np.linspace(-8, 8, 401), 0.2
p, q = [np.exp(-(x - c) ** 2 / 2) / np.sqrt(2 * np.pi) for c in (0, mu)]
for gamma in (0, 0.01, 1, 10):
    large = mu ** 2 / (16 * gamma) if gamma else np.inf
    print(f'gamma {gamma:>5}: exact {penalized_value(p, q, gamma, x):.3e}, '
          f'local {mu ** 2 / (4 + 16 * gamma):.3e}, large-gamma {large:.3e}')
```

```
gamma     0: exact 9.950e-03, local 1.000e-02, large-gamma inf
gamma  0.01: exact 9.566e-03, local 9.615e-03, large-gamma 2.500e-01
gamma     1: exact 1.996e-03, local 2.000e-03, large-gamma 2.500e-03
gamma    10: exact 2.438e-04, local 2.439e-04, large-gamma 2.500e-04
```

The local formula matches the exact value to within one percent at every γ. The large-γ form holds only when the penalty dominates: at $\gamma = 0.01$ it gives 0.25 against the exact 0.0096, 26 times too large, and at $\gamma = 10$ it is within three percent.

16.4.3.3 A Low-Pass Filter

A general $p - q$ has many length scales. Where m is roughly constant, $\mathcal{L}_m$ multiplies a Fourier component $e^{i\omega x}$ by $m\omega^2$, so in Equation 16.31 the component with wavenumber ω enters with weight

$$\frac{1}{|\ell''(0)| + \gamma\omega^2} \qquad \text{instead of} \qquad \frac{1}{|\ell''(0)|}.$$

Components whose length scale $1/\omega$ exceeds $\sqrt{\gamma/|\ell''(0)|}$, which is $2\sqrt{\gamma}$ for the logistic loss, keep nearly their unpenalized weight; shorter ones are damped by about the factor $|\ell''(0)|/(\gamma\omega^2)$. The penalty is a low-pass filter on $p - q$ with cutoff length $2\sqrt{\gamma}$. Above the cutoff the game measures 2 JS; below it, the dual Sobolev norm of Equation 16.32, which equals $W_2(p, q)$ to leading order as $q \to p$ by the Benamou–Brenier form of optimal transport (Section 38.6.3). Below the cutoff the generator therefore keeps a signal when nearby supports move apart. Because the critic's input gradient has units of one over length, γ is

a squared length, set by the scale of the data: R3GAN uses values from 0.05 on CIFAR-10 to 150 on FFHQ at 256×256 pixels.

16.4.4 The R3GAN Recipe

R3GAN (Huang et al. 2024) puts the pieces together. Its critic minimizes the pairing loss of Section 16.3 plus both zero-centered penalties, and its generator minimizes the non-saturating form of that loss. The penalties make the equilibrium locally attracting; the pairing contributed the coverage measured in the StackedMNIST ablation above. With this loss R3GAN trains a plain convolutional network without most of the stabilizers that earlier GANs needed (Section 16.5).

16.4.4.1 The Loss in Code

The implementation keeps the losses and the penalties apart. For aligned pairs of a real sample x and a generated sample x', the critic minimizes $\text{softplus}(D(x') - D(x)) = -\log \sigma(D(x) - D(x'))$ and the generator the reversed $\text{softplus}(D(x) - D(x'))$. The functions `gan_loss_D` and `gan_loss_G` give the logistic critic loss and the non-saturating generator loss of Section 16.1 with the same arguments, so a training loop can take either pair.

```
def rpgan_loss_D(critic, real, fake):
    """Relativistic pairing critic loss -E[log sigma(D(x) - D(x'))]."""
    return F.softplus(critic(fake) - critic(real)).mean()


def rpgan_loss_G(critic, real, fake):
    """Non-saturating pairing loss for the generator."""
    return F.softplus(critic(real) - critic(fake)).mean()


def gan_loss_D(critic, real, fake):
    """Logistic critic loss -E[log sigma(D(x))] - E[log sigma(-D(x'))]: the
    batch means, where update_D reports the halved sum."""
    return F.softplus(-critic(real)).mean() + F.softplus(critic(fake)).mean()


def gan_loss_G(critic, real, fake):
    """Non-saturating generator loss -E[log sigma(D(x'))]; real is unused."""
    return F.softplus(-critic(fake)).mean()
```

The penalty function returns the per-sample squared input gradients on both batches, before the factor $\gamma/2$. It differentiates the *sum* of the critic's scores with respect to the input batch: each score depends only on its own row, so one backward pass gives every sample's input gradient (batch normalization, which mixes rows, would break this). The gradient stays differentiable, so that training can differentiate the penalty with respect to the critic's parameters; the samples are constants. The critic may be any function from a batch of inputs to a batch of scores, including a closure that fixes a condition (Section 16.6).

`create_graph=True` keeps the graph of the input gradient for the second differentiation, and `detach` makes the samples constants.

```
def r1_r2_penalty(critic, real, fake):
    """Per-sample squared critic input gradients on real (R1) and fake (R2),
    before the gamma/2 scale."""
    def sq_grad_norm(x):
        x = x.detach().requires_grad_(True)
        grad, = torch.autograd.grad(critic(x).sum(), x, create_graph=True)
        return grad.reshape(x.shape[0], -1).pow(2).sum(dim=1)
    return sq_grad_norm(real), sq_grad_norm(fake)
```

16.4.4.2 The Training Steps

A training loop alternates one step of each player. `critic_step` takes one optimizer step on the critic loss plus $\gamma/2$ times both penalties, with the generated batch held fixed, and returns the loss and the batch means of R_1 and R_2, so that a loop can log the critic's steepness. `generator_step` then takes one step on the generator loss against the updated critic. Given labels `y`, both score with the conditional critic $D(\cdot, y)$ (Section 16.6).

```
def critic_step(net_D, opt_D, loss_D, real, fake, gamma, y=None):
    """One critic step on loss_D + gamma/2 (R1 + R2); return the loss and
    the batch means of R1 and R2."""
    critic = net_D if y is None else lambda x: net_D(x, y)
    fake = fake.detach()  # the generator is held fixed
    r1, r2 = r1_r2_penalty(critic, real, fake)
    loss = loss_D(critic, real, fake)
    opt_D.zero_grad()
    (loss + gamma / 2 * (r1 + r2).mean()).backward()
    opt_D.step()
    return loss.detach(), r1.mean().detach(), r2.mean().detach()


def generator_step(net_G, opt_G, loss_G, net_D, real, z, y=None):
    """One generator step on loss_G against the critic; return the loss."""
    critic = net_D if y is None else lambda x: net_D(x, y)
    loss = loss_G(critic, real, net_G(z) if y is None else net_G(z, y))
    opt_G.zero_grad()
    loss.backward()
    opt_G.step()
    return loss.detach()
```

16.4.4.3 Mode Coverage on 25 Gaussians

We test the recipe on a standard problem with many modes: 25 Gaussians whose centers form a 5×5 grid with spacing 2, each with standard deviation 0.05. A generated sample is *on a mode* if it lies within three widths, 0.15, of the nearest center, where a 2-D Gaussian puts $1 - e^{-9/2} \approx 98.9\%$ of its mass. For 10,000 samples we report how many modes are covered, counting a mode only if it holds at least a fifth of the on-mode samples it would receive if they were spread evenly, then the fraction of samples on a mode, their mean distance to the nearest center (about 0.06 for a perfect fit), and $\mathbb{E}_q \|\nabla_x D\|^2$ for the final critic. Both networks are MLPs with two hidden layers of 256 units, built by `d21.make_mlp` of Section 16.1; the generator maps a 64-dimensional latent vector to the plane, and the critic uses leaky

ReLU and no normalization.

```
def grid_centers(side=5, spacing=2.0):
    coords = (torch.arange(side, dtype=torch.float32) - (side - 1) / 2)
    gx, gy = torch.meshgrid(coords * spacing, coords * spacing, indexing='ij')
    return torch.stack([gx.reshape(-1), gy.reshape(-1)], dim=1)

def sample_modes(centers, n, width=0.05):
    idx = torch.randint(0, centers.shape[0], (n,), device=centers.device)
    return centers[idx] + width * torch.randn(n, 2, device=centers.device)
```

The training function takes the two losses and γ as arguments and alternates `critic_step` and `generator_step`, each on a fresh batch. Both networks use Adam with $\beta_1 = 0$ and $\beta_2 = 0.99$, as R3GAN does, and a learning rate that decays from $2 \cdot 10^{-3}$ to $2 \cdot 10^{-4}$ along a cosine over 10,000 steps with batches of 1,024. It returns the generator and $\mathbb{E}_q \|\nabla_x D\|^2$ from the last critic step, which at $\gamma = 0$ is the only use of the penalties.

```
def train_grid(centers, loss_D, loss_G, gamma, steps=10000, lr=2e-3,
               batch=1024, latent=64, seed=0):
    torch.manual_seed(seed)
    dev = centers.device
    net_G = d2l.make_mlp([latent, 256, 256, 2], nn.ReLU()).to(dev)
    net_D = d2l.make_mlp([2, 256, 256, 1], nn.LeakyReLU(0.2)).to(dev)
    opt_G, opt_D = [torch.optim.Adam(net.parameters(), lr, betas=(0.0, 0.99))
                    for net in (net_G, net_D)]
    scheds = [torch.optim.lr_scheduler.CosineAnnealingLR(opt, steps, lr / 10)
              for opt in (opt_G, opt_D)]
    for _ in range(steps):
        real = sample_modes(centers, batch)
        with torch.no_grad():
            fake = net_G(torch.randn(batch, latent, device=dev))
        _, _, grad_q = critic_step(net_D, opt_D, loss_D, real, fake, gamma)
        z = torch.randn(batch, latent, device=dev)
        generator_step(net_G, opt_G, loss_G, net_D,
                       sample_modes(centers, batch), z)
        for sched in scheds:
            sched.step()
    return net_G, grad_q.item()           # E_q ||grad_x D||^2
```

We train three runs from the same initialization: the classical GAN without a penalty, and the R3GAN loss at $\gamma = 1$ and at $\gamma = 10$. The low-pass filter predicts how the last two differ. The grid repeats every 2 units, a component with wavenumber $\omega = \pi$, which the penalty damps about 40-fold at $\gamma = 1$ and about 400-fold at $\gamma = 10$. With a ten times weaker signal about where the modes sit, the $\gamma = 10$ generator should resolve the grid less sharply within the same budget.

```
centers = grid_centers().to(d2l.try_gpu())
arms = {'GAN': (gan_loss_D, gan_loss_G, 0.0),
        'R3GAN, gamma=1': (rpgan_loss_D, rpgan_loss_G, 1.0),
        'R3GAN, gamma=10': (rpgan_loss_D, rpgan_loss_G, 10.0)}
latent, samples, grad_q = 64, {}, {}
```

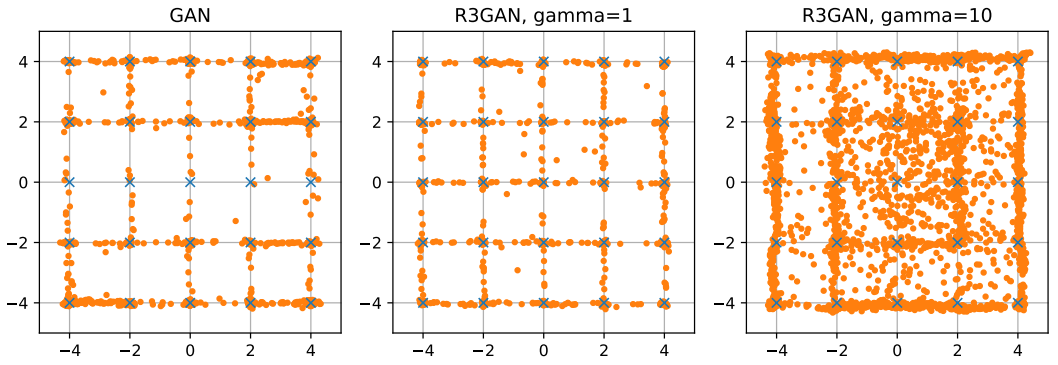
```
for name, (loss_D, loss_G, gamma) in arms.items():
    net_G, grad_q[name] = train_grid(centers, loss_D, loss_G, gamma,
                                     latent=latent)
    with torch.no_grad():
        z = torch.randn(10000, latent, device=centers.device)
        samples[name] = net_G(z).cpu().numpy()
grid = centers.cpu().numpy()
```

Each panel shows 3,000 generated samples in orange and the 25 centers in blue.

```
def mode_stats(x, centers, width=0.05):
    """Modes covered (each holds a fifth of an even share of the on-mode
    samples), fraction within three widths, mean distance."""
    dist = np.linalg.norm(x[:, None] - centers[None], axis=-1)
    near, idx = dist.min(axis=1), dist.argmin(axis=1)
    on = near <= 3 * width
    counts = np.bincount(idx[on], minlength=len(centers))
    covered = (counts >= on.sum() / len(centers) / 5).sum()
    return covered, on.mean(), near.mean()

fig, axes = d2l.plt.subplots(1, 3, figsize=(9.5, 3.4))
for ax, (name, x) in zip(axes, samples.items()):
    modes, on, dist = mode_stats(x, grid)
    print(f'{name}: {modes}/25 modes, on-mode {on:.2f}, '
          f'distance {dist:.2f}, E_q|grad D|^2 {grad_q[name]:.1e}')
    # Centers drawn last, on top of the samples
    d2l.plot([x[:3000, 0], grid[:, 0]], [x[:3000, 1], grid[:, 1]],
             fmts=('C1.', 'C0x'), xlim=(-5, 5), ylim=(-5, 5), axes=ax)
    ax.set_title(name)
    ax.set_aspect('equal')
fig.tight_layout()
```

```
GAN: 20/25 modes, on-mode 0.85, distance 0.12, E_q|grad D|^2 2.9e+00
R3GAN, gamma=1: 25/25 modes, on-mode 0.88, distance 0.11, E_q|grad D|^2 2.5e-04
R3GAN, gamma=10: 24/25 modes, on-mode 0.17, distance 0.49, E_q|grad D|^2 1.4e-04
```



The R3GAN loss at $\gamma = 1$ fits the grid. Over sixteen runs with different seeds it covered all 25 modes in every run but one, which covered 24, and put most of its samples within three widths of a center. The classical GAN also puts most of its samples on a mode but spreads

them less evenly. Over about 150 runs it covered all 25 modes in only about two runs out of five, and in more than a quarter of them it left a whole row or column of the grid nearly empty. The two losses differ more reliably in the critic. With the penalty, $\mathbb{E}_q\|\nabla_x D\|^2$ ends about three to four orders of magnitude below its value for the unpenalized critic, which stays steep at the generated samples even when the fit is good, like the undamped slope ψ of the Dirac-GAN. At $\gamma = 10$ the grid is blurred, as predicted: only an eighth to three fifths of the samples lie on a mode, and their mean distance to the nearest center is 1.5 to 6 times that at $\gamma = 1$. The blur also thins out some modes, mostly near the center of the grid, so the strict count finds between 21 and 25 covered. The equilibrium is still $q = p$; the larger γ slows the approach to it (exercise 4).

16.4.5 The Games of This Chapter

16.4.5.1 Three Choices

Four sections have now derived what each adversarial game measures. Three choices describe all of them (Table 16.3): what the critic estimates, what it does with the estimate, and how the critic is constrained. The last column records whether the generator keeps a gradient when the supports of p and q are disjoint.

Table 16.3: The games of this chapter, described by three choices: what the critic estimates, what it does with the estimate, and how the critic is constrained. The last column records whether the generator keeps a gradient on disjoint supports.

objective	the critic estimates	what it does with the estimate	how the critic is constrained	on disjoint supports
log loss, f-GAN (Section 16.1, Section 16.2)	$\log(p/q)$, or $f'(p/q)$ for an f-GAN	classifies one sample	not at all	saturates
MMD (Section 16.2)	a kernel-smoothed $p - q$, the witness	witnesses a difference of means	a fixed kernel ball	smooth
Wasserstein-1 (Section 16.2)	a Kantorovich potential	witnesses a difference of means	a Lipschitz ball	smooth
pairing, RpGAN (Section 16.3)	$\log(p/q)$ up to a constant	ranks a real–generated pair	not at all	saturates
RpGAN + R_1 + R_2 (Section 16.4)	$\log(p/q)$ above the length scale $2\sqrt{\gamma}$, smoothed below	ranks a real–generated pair	a symmetric Sobolev penalty	smooth, near equilibrium

16.4.5.2 Reading the Table

The classifying and the ranking critics estimate the same log ratio; the loss and the pairing change how its errors are weighted, not what is estimated. The MMD and W_1 critics estimate a smoothed difference and a potential, and the penalty moves the pairing critic toward them below the length scale $2\sqrt{\gamma}$. The constraint decides whether a gradient survives on

disjoint supports, and the zero-centered penalty also damps the rotation of the Dirac-GAN. Ranking guarantees coverage only for a fixed coupling; for minibatch training the evidence is the StackedMNIST ablation, where the pairing loss recovered all 1,000 modes and the classical loss 693.

Whether this recipe trains an image generator, and how to score the images it produces, is the question of Section 16.5.

16.4.6 Summary

Gradient training of an adversarial game need not reach its equilibrium. On the Dirac-GAN the gradient field is orthogonal to the parameters, so the flow circles the solution and every simultaneous step moves away from it, whatever the loss, its weighting or its pairing. The zero-centered penalties R_1 and R_2 on the critic's input gradient at data and generated samples damp the rotation for every $\gamma > 0$. Near equilibrium the penalized game measures $p - q$ through a low-pass filter with cutoff length $2\sqrt{\gamma}$: 2 JS at larger scales, a linearized Wasserstein-2 distance at smaller ones. R3GAN combines the paired loss with both penalties; on 25 Gaussians it fits the grid at $\gamma = 1$ with a critic that stays flat on generated samples, while $\gamma = 10$ blurs the grid within the same budget.

16.4.7 Exercises

1. code **Discrete-time stability.** A simultaneous step of size h maps (θ, ψ) to $(\theta, \psi) + h\,\mathbf{u}(\theta, \psi)$, with $\mathbf{u}$ from Equation 16.26 plus $-\gamma\psi$ in its second component for a penalized critic.
    1. Show that at $\gamma = 0$ the Jacobian of the map at the equilibrium has eigenvalues $1 \pm i\,h\,\ell'(0)$, so no step size makes the unpenalized game locally convergent.
    2. For $\gamma > 0$ the eigenvalues are $1 + h\nu$, with ν from Equation 16.30. Show that in the underdamped regime $\gamma < 2\ell'(0)$ the iteration converges locally if and only if $h < \gamma/\ell'(0)^2$.
    3. Check the threshold of (b) at $\gamma = 0.3$: run `trajectory((0.1, 0.1), h, 3000, gamma=0.3)` for $h = 1.1$ and $h = 1.3$ and plot the distance from the equilibrium. What does the iteration do above the threshold?
2. **Penalized best response.** Let the critic of the Dirac-GAN subtract both penalties of Equation 16.28, which for the linear critic sum to $\gamma\psi^2$, take $\ell = \log\sigma$, and write $V_\gamma(\theta) = \sup_\psi \{V(\theta, \psi) - \gamma\psi^2\}$.
    1. Show that the supremum is attained for every θ, and that for small θ the maximizer is $\psi^\star \approx -\ell'(0)\,\theta/(2\gamma)$ and $V_\gamma(\theta) = 2\ell(0) + \frac{\ell'(0)^2}{4\gamma}\,\theta^2 + O(\theta^4)$.
    2. Compare (a) with Equation 16.32, using $W_2(\delta_0, \delta_\theta) = |\theta|$. Why does the curvature term of Equation 16.31 not appear at order θ^2? What gradient does the generator receive near $\theta = 0$, and how does it depend on γ?
    3. Now let the critic be any differentiable function, penalized by $\gamma \int m\,\|\nabla_x D\|^2$. Show that the penalized best-response value is $2\sup_t \ell(t)$ for every $\theta \neq 0$. Which assumption behind Equation 16.31 fails here?
3. **Penalizing one side only.** Let $p = \delta_0$, put the generated point at $x' = 1$, take $\ell = \log\sigma$, and consider the critics $D_c(x) = -c\,x^2$ with $c \geq 0$, flat at the data and steep at the generated point.

1. Evaluate R_1 and R_2 for D_c. Show that the critic objective $V_\ell(D_c) - R_1$ increases with c and has no maximizer.
2. Show that the gradient of the non-saturating generator loss $-\log\sigma(D(x'))$ with respect to x' at $x' = 1$ grows without bound in c. What does the saturating loss $\log\sigma(-D(x'))$ give instead? Which observation of Huang et al. (2024) does this reproduce?
3. Subtract $R_1 + R_2$ instead. Show that the maximizing $c^\star$ is finite with $c^\star < 1/(8\gamma)$, and bound the generator's gradient at $c^\star$.

4. code **The penalty weight as a length.** In Section 16.4.4.3, $\gamma = 10$ blurs the grid that $\gamma = 1$ fits.
   1. For data scaled by $c > 0$ and the critic $D_c(x) = D(x/c)$, show that the penalized game at weight $c^2\gamma$ has the same value as the original game at weight γ.
   2. With the low-pass weight and $|\ell''(0)| = \frac{1}{4}$, compute by what factor the penalty damps a component whose wavelength is the mode width 0.05, at $\gamma = 1$ and at $\gamma = 10$. Compare the mean distance printed for $\gamma = 1$ with the 0.06 of a perfect fit. Does the damping predict how sharply each mode is resolved?
   3. Train the $\gamma = 10$ run for two and four times as many steps, with the cosine schedule stretched to match. Does the on-mode fraction approach that of $\gamma = 1$?

5. code **Alternating updates.** ⊠ `train_grid` updates the critic first and then the generator against the updated critic. On the Dirac-GAN this reads

$$\psi_{k+1} = \psi_k + h\,u_\psi(\theta_k, \psi_k), \qquad \theta_{k+1} = \theta_k + h\,u_\theta(\theta_k, \psi_{k+1}), \tag{16.33}$$

   with (u_θ, u_ψ) the field of Equation 16.26.
   1. Linearize at the equilibrium for $\gamma = 0$ and show that the Jacobian of the combined update has determinant one, with eigenvalues on the unit circle if and only if $h\,\ell'(0) \le 2$. What does this imply about linear convergence and divergence?
   2. Write an alternating `trajectory` and run it at $\gamma = 0$ from $(0.1, 0.1)$ for 10^5 steps with $h \in \{0.2, 1, 3, 5\}$. Record the smallest and largest distance from the equilibrium over the last 10^4 steps, and explain the wide spread at $h = 3$. Repeat at $h = 0.2$ with $\gamma = 0.3$.

6. code **The local value.** ⊠
   1. Perturb the maximizing critic of Equation 16.31 to $D + s\,\xi$ with a test function ξ and show that the first-order condition in s is $(|\ell''(0)|\,m + \gamma\mathcal{L}_m)D = \frac{\ell'(0)}{2}(p - q)$, assuming that boundary terms vanish.
   2. Let $p = \mathcal{N}(0, 1)$ and $q = p\,(1 + 0.1\sin\omega x)$. Use `penalized_value` to compute the ratio of the values at γ and at $\gamma = 0$ for $\omega \in \{0.5, 2, 8\}$ and $\gamma \in \{0.01, 0.1, 1\}$, and compare it with the low-pass weight $1/(1 + 4\gamma\omega^2)$. For which ω does the prediction fail, and which assumption of the Fourier argument breaks there?

7. code **One-centered penalty dynamics.** ⊠ Take the linear score $\ell(t) = t$ of the Wasserstein critic, so $\ell'(0) = 1$. On the Dirac-GAN the one-centered penalty becomes $\frac{\gamma}{2}(|\psi| - 1)^2$, which the critic subtracts.
   1. Show that the simultaneous gradient field vanishes at no point with $\psi \neq 0$, and find where its one-sided limits as $\psi \to 0^\pm$ vanish. Where is the generator there?

2. On $\psi > 0$ the field is affine. Compare its Jacobian with J_γ of Equation 16.30; for which γ are its eigenvalues real?
3. Modify `dirac_field` and run `trajectory` from $(1, 1)$ with $h = 0.01$ for 40,000 steps at $\gamma = 1$ and $\gamma = 3$. Where does each trajectory end? Explain both outcomes with (a) and (b).

8. code **Momentum in a rotating game.** R3GAN trains with Adam at $\beta_1 = 0$. To isolate the first moment, average the field of the penalized Dirac-GAN with the logistic loss, $\bar{\mathbf{u}}_k = \beta\, \bar{\mathbf{u}}_{k-1} + (1-\beta)\, \mathbf{u}(\mathbf{x}_k)$ and $\mathbf{x}_{k+1} = \mathbf{x}_k + h\, \bar{\mathbf{u}}_k$, with $\mathbf{x}_k = (\theta_k, \psi_k)$ and $\bar{\mathbf{u}}_{-1} = \mathbf{0}$. Linearize at the equilibrium, write one step as a linear map of $(\mathbf{x}_k, \bar{\mathbf{u}}_{k-1}) \in \mathbb{R}^4$, and compute numerically the largest stable step size for $\beta \in \{0, 0.5, 0.9\}$ at $\gamma = 0.3$ and $\gamma = 1$. Check $\beta = 0$ against problem 1b. In which damping regime does momentum shrink the range of stable step sizes?

9. code **Non-saturating Dirac dynamics.** The non-saturating generator replaces $\ell'(-\psi\theta)$ by $\ell'(\psi\theta)$ in the first component of Equation 16.26. Take $\ell = \log\sigma$, $\gamma = 0$ and $r^2 = \theta^2 + \psi^2$.
   1. Show that $\frac{d}{dt} r^2 = 2\psi\theta\,(\ell'(\psi\theta) - \ell'(-\psi\theta)) \le 0$ for a concave ℓ. Why can the approach not be linear?
   2. Average $\frac{d}{dt} r^2$ over one turn, at angular rate $\ell'(0)$, to leading order in r, and show that $r(t)^2 \approx 8/t$ for large t.
   3. One simultaneous step changes r^2 by $2h\,\langle \mathbf{x}, \mathbf{u}\rangle + h^2\|\mathbf{u}\|^2$. Average this over one turn and show that it vanishes at $(r^\star)^2 = 2h$. Is this cycle stable?
   4. Redefine `dirac_field` with the non-saturating weight, run `trajectory` from $(1, 1)$ and from $(0.05, 0.05)$ for $4000/h$ steps with $h \in \{0.01, 0.05, 0.2\}$, and compare the late distance with $r^\star$. Check (b) with `flow=True`.

16.5 Adversarial Image Generation

The multilayer perceptron of Section 16.1 learned to draw Fashion-MNIST garments in a few minutes on a CPU. Its shirts, dresses and shoes were recognizable but grainy, and the game never settled: the critic's loss stayed well below its equilibrium value. That network treats each pixel as an input of its own and knows nothing of which pixels are neighbors. A color image of 64×64 pixels has $12{,}288$ values, all of which the generator must produce from about a hundred random numbers.

Convolutional networks made adversarial training work on images. In 2015 a convolutional generator trained on photographs of bedrooms produced bedrooms that no camera had taken (Radford et al. 2015), and within a few years generated faces were hard to tell from photographs. Images also raise a second problem: a grid of samples reveals a collapse, but two good generators look alike to the eye.

This section takes up both problems on 40,000 small game sprites. We first build the 2015 network, which trained the original loss on images through its architecture alone. We then build a network without normalization layers and train it three times, changing one ingredient at a time: the classic loss, the classic loss with the two gradient penalties of Section 16.4, and the pairing loss of Section 16.3 with the same penalties. To compare the runs we need a number; the two standard scores turn out to be distances from Section 16.2, computed in the feature space of an image classifier.

```
import torchvision
```

16.5.1 The 2015 Recipe

The deep convolutional GAN (DCGAN) of Radford et al. (2015) kept the original objective and changed the networks. Its generator grows an image with transposed convolutions, and its discriminator shrinks one with strided convolutions. The generator uses ReLU and ends in a tanh, which matches its range to images scaled to $[-1, 1]$. The discriminator uses the leaky ReLU, which passes $0.2x$ for negative x, so that no unit stops passing gradient to the generator. Adam runs with $\beta_1 = 0.5$ instead of the usual 0.9, which shortens the optimizer's memory of a gradient field that changes as the other player moves.

16.5.1.1 The Sprite Dataset

The dataset contains 40,597 sprite images of 721 Pokemon species, with many variants of each, obtained from pokemondb. Sprites are small, centered on white backgrounds, and diverse in silhouette and palette. We resize each image to 64×64 and scale its pixel values to $[-1, 1]$. The whole dataset fits in memory, so `load_pokemon` decodes it once into one array.

```
d2l.DATA_HUB['pokemon'] = (d2l.DATA_URL + 'pokemon.zip',
                           'c065c0e2593b8b161a2d7873e42418bf6a21106c')


def load_pokemon(resolution=64):
    """All Pokemon sprites as one (n, 3, h, w) tensor in [-1, 1]."""
    import warnings  # not in the d2l library preamble
    transform = torchvision.transforms.Compose([
        torchvision.transforms.Resize((resolution, resolution)),
        torchvision.transforms.PILToTensor()])
    dataset = torchvision.datasets.ImageFolder(
        d2l.download_extract('pokemon'), transform=transform)
    loader = torch.utils.data.DataLoader(
        dataset, batch_size=512, num_workers=d2l.get_dataloader_workers())
    with warnings.catch_warnings():  # palette sprites with transparency
        warnings.filterwarnings('ignore', message='Palette images')
        return torch.cat([X for X, _ in loader]).float() / 127.5 - 1

images = load_pokemon()
images.shape
```

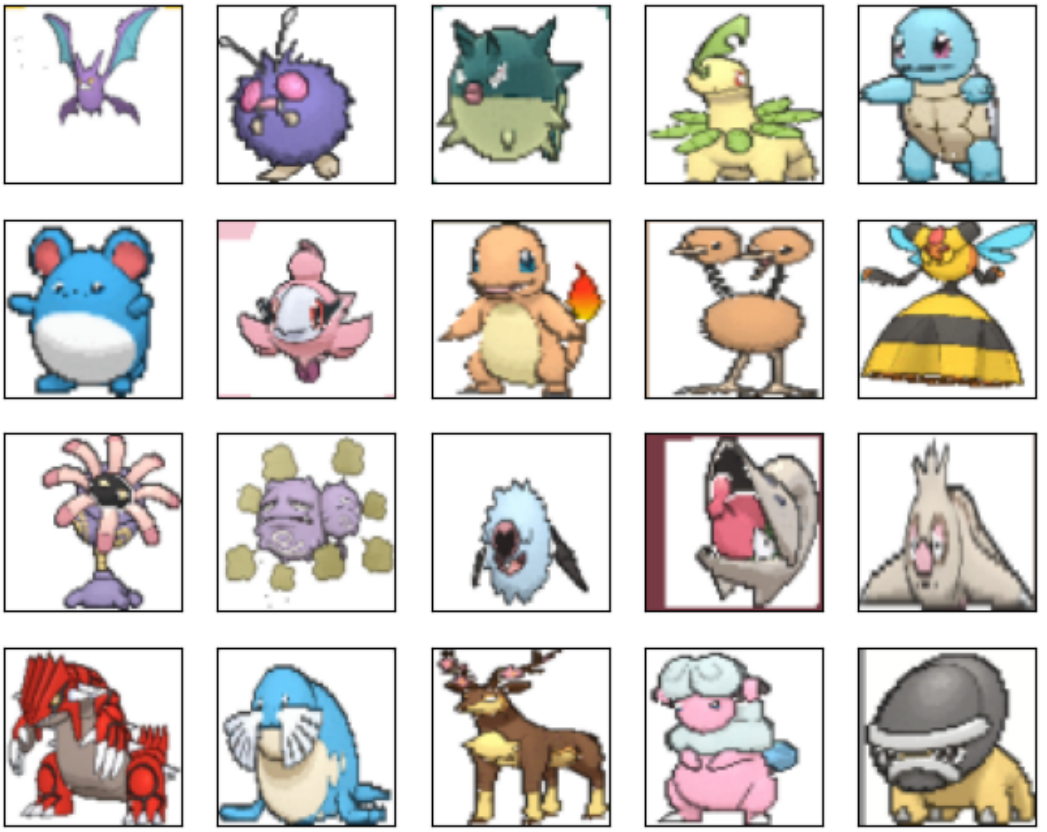
```
torch.Size([40597, 3, 64, 64])
```

A fixed permutation sets aside one tenth of the images, which are never used for training. They are the real sample against which generated images are scored, and they show whether the critic scores its training images higher than unseen ones. The split is by image, so variants of one species can fall on both sides.

```
device = d2l.try_gpu()
perm = np.random.default_rng(12345).permutation(len(images))
n_holdout = len(images) // 10
```

```
train_imgs = images[perm[n_holdout:]].to(device)
holdout_imgs = images[perm[:n_holdout]].to(device)
print(f'{len(train_imgs)} training and {len(holdout_imgs)} held-out images')
d2l.show_images(train_imgs[:20].cpu().permute(0, 2, 3, 1) / 2 + 0.5,
                num_rows=4, num_cols=5);
```

```
36538 training and 4059 held-out images
```



16.5.1.2 Transposed Convolutions and Batch Normalization

The generator must grow a 1×1 input with 100 channels into a 64×64 image. A *transposed convolution* grows a feature map by running a convolution's arithmetic backwards. Each input pixel stamps a copy of the kernel, scaled by the pixel's value, onto the output; successive stamps start one stride apart, and where they overlap their values add. In one dimension, the input $(1, 2)$ and a kernel of four ones with stride 2 give the stamps $(1, 1, 1, 1)$ and $(2, 2, 2, 2)$, two positions apart, so the output is $(1, 1, 3, 3, 2, 2)$. If pad rows and columns are trimmed from each border, an $n \times n$ input gives an output of side

$$\text{stride} \cdot (n - 1) + \text{kernel} - 2 \cdot \text{pad}. \tag{16.34}$$

With kernel 4, stride 2, and padding 1 the side is $2n$, so each such layer doubles the resolution; with stride 1 and no padding the same kernel turns a 1×1 input into a 4×4 map. A

strided convolution with the same settings halves the side.

The DCGAN generator chains five transposed convolutions through the resolutions $1 \to 4 \to 8 \to 16 \to 32 \to 64$, halving the number of channels at each doubling. The discriminator mirrors it with strided convolutions down to 4×4 and a last 4×4 convolution that returns one logit (Figure 16.6). Each block is a convolution, batch normalization, and an activation. Batch normalization keeps the scale of the activations in range while the weights change; following Radford et al., we leave it out of the discriminator's input block and the generator's output layer.

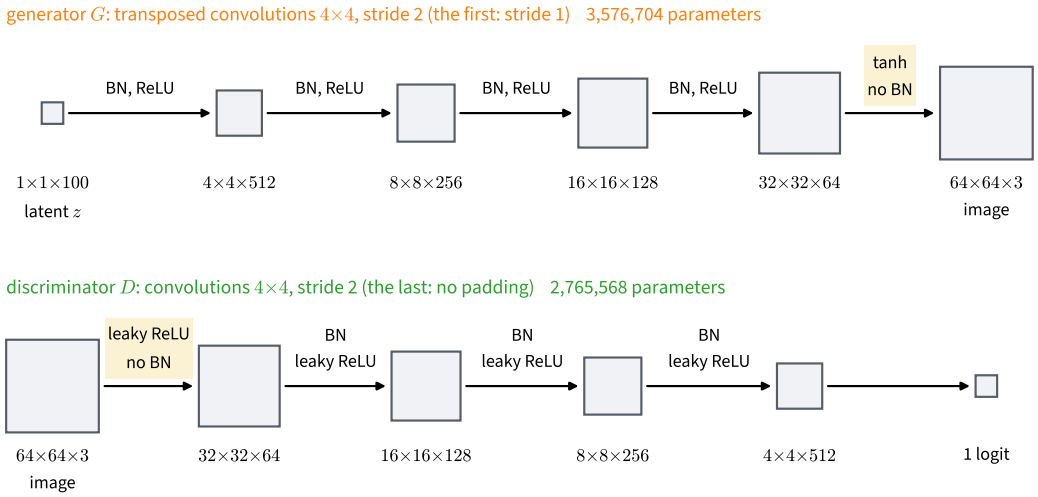


Figure 16.6: The DCGAN architecture (Radford et al. 2015). The generator maps a 100-dimensional latent vector through transposed 4×4 convolutions to feature maps of side $4, 8, 16, 32$ and a 64×64 RGB image. The discriminator reverses the schedule with strided 4×4 convolutions down to one logit. Every block uses batch normalization except the two shaded ones: the generator's output layer (tanh) and the discriminator's input block (leaky ReLU).

```
def G_block(c_in, c_out, stride=2, padding=1):
    """Transposed convolution (kernel 4), batch norm, ReLU."""
    return nn.Sequential(
        nn.ConvTranspose2d(c_in, c_out, 4, stride, padding, bias=False),
        nn.BatchNorm2d(c_out), nn.ReLU())

def D_block(c_in, c_out, norm=True):
    """Strided convolution (kernel 4), batch norm, leaky ReLU."""
    return nn.Sequential(
        nn.Conv2d(c_in, c_out, 4, 2, 1, bias=False),
        nn.BatchNorm2d(c_out) if norm else nn.Identity(), nn.LeakyReLU(0.2))
```

```
torch.manual_seed(0)
net_G = nn.Sequential(                                  # (100,)
    nn.Unflatten(1, (100, 1, 1)),                       # (100, 1, 1)
```

```
    G_block(100, 512, stride=1, padding=0),              # (512, 4, 4)
    G_block(512, 256), G_block(256, 128), G_block(128, 64),  # (64, 32, 32)
    nn.ConvTranspose2d(64, 3, 4, 2, 1, bias=False), nn.Tanh()).to(device)
net_D = nn.Sequential(                                   # (3, 64, 64)
    D_block(3, 64, norm=False),                          # (64, 32, 32)
    D_block(64, 128), D_block(128, 256), D_block(256, 512),  # (512, 4, 4)
    nn.Conv2d(512, 1, 4, bias=False)).to(device)         # (1, 1, 1)
X = net_G(torch.zeros(2, 100, device=device))
print(X.shape, net_D(X).shape)
```

```
torch.Size([2, 3, 64, 64]) torch.Size([2, 1, 1, 1])
```

16.5.1.3 Training with the Classic Loss

The loop `d2l.train_gan` of Section 16.1, which alternates a logistic-regression step of the critic with a non-saturating step of the generator over shuffled minibatches, applies to images unchanged. We use the DCGAN settings: minibatches of 256 images with the incomplete last one dropped, learning rate $2 \cdot 10^{-4}$, $\beta_1 = 0.5$, convolution kernels drawn from $\mathcal{N}(0, 0.02^2)$, batch-normalization scales from $\mathcal{N}(1, 0.02^2)$, and offsets at zero.

```
def dcgan_init(module):
    if isinstance(module, (nn.Conv2d, nn.ConvTranspose2d)):
        nn.init.normal_(module.weight, 0, 0.02)
    elif isinstance(module, nn.BatchNorm2d):
        nn.init.normal_(module.weight, 1, 0.02)
        nn.init.zeros_(module.bias)

for net in (net_D, net_G):
    net.apply(dcgan_init)
dataset = torch.utils.data.TensorDataset(train_imgs)
# Sample whole minibatches of indices, so that the GPU tensor is indexed
# once per minibatch rather than once per image
sampler = torch.utils.data.BatchSampler(
    torch.utils.data.RandomSampler(dataset), 256, drop_last=True)
data_iter = torch.utils.data.DataLoader(dataset, sampler=sampler,
                                        batch_size=None)
```

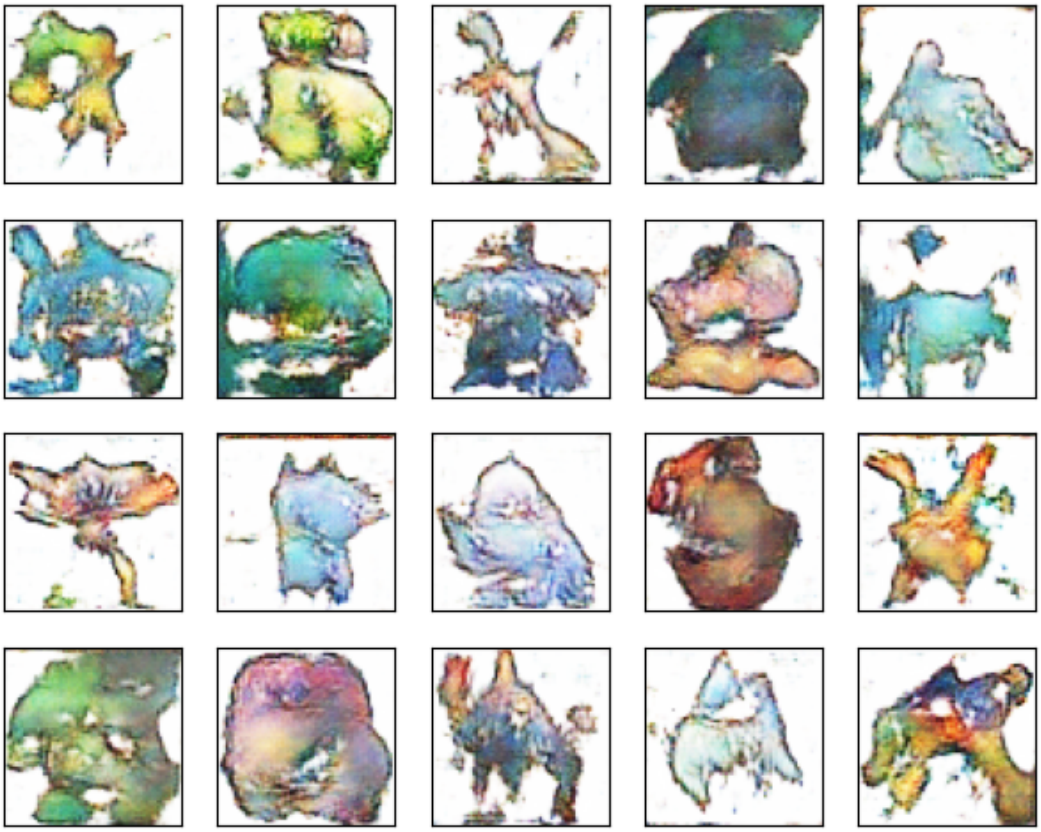
Twenty epochs take a minute or two on one GPU. The printed losses are per image; at the equilibrium both would equal $\log 2 \approx 0.69$.

```
history = d2l.train_gan(net_D, net_G, data_iter, num_epochs=20,
                        latent_dim=100, lr_D=0.0002, lr_G=0.0002, seed=1)
loss_D, loss_G = history[-1]
print(f'last epoch: loss_D {loss_D:.3f}, loss_G {loss_G:.3f}')
```

```
last epoch: loss_D 0.241, loss_G 2.940
```

We sample in evaluation mode, in which batch normalization uses the statistics collected during training, so that each image depends only on its own latent code.

```
net_G.eval()
with torch.no_grad():
    fake = net_G(torch.randn(20, 100, device=device))
d2l.show_images(fake.cpu().permute(0, 2, 3, 1) / 2 + 0.5,
                num_rows=4, num_cols=5);
```



The samples are diverse: centered, creature-like shapes on mostly white backgrounds, with plausible palettes and blotchy textures. As in the garment run, the game is unbalanced, and here it is further from equilibrium: the critic's loss is far below $\log 2$ and the generator's several times above it. Yet the generator does not collapse onto one output. The architecture, not a change of loss, keeps this game trainable.

16.5.2 A Backbone without Normalization

16.5.2.1 Six Principles

For several years after DCGAN, most progress on image GANs came from architectural devices, each added to repair one failure of training. R3GAN (Huang et al. 2024) asked whether a well-behaved loss makes them unnecessary. It lists six differences between its minimal baseline and DCGAN: (a) a convergent objective, in its experiments the pairing loss with both zero-centered penalties of Section 16.4; (b) a small learning rate and Adam without first-moment momentum, $\beta_1 = 0$; (c) no normalization layer in either network; (d) bilinear

resampling instead of strided or transposed convolutions; (e) leaky ReLU in both networks and no tanh at the generator's output; and (f) a 4×4 constant input, output skips, and a residual critic. Violating (a), (b), or (c) often makes training fail, while violating (d) or (e) lowers sample quality. StyleGAN2 (Karras, Laine, et al. 2020), the strongest face generator of 2020, relied on about ten such devices; rebuilt from these principles without them, a network of about its size scored slightly better on photographs of faces.

16.5.2.2 The Backbone

We build a small network that follows principles (b) to (e) at sprite scale (Figure 16.7). Both networks resample with bilinear interpolation followed by an ordinary 3×3 convolution, which avoids the uneven overlap of transposed-convolution stamps that produces checkerboard artifacts (Odena et al. 2016). Neither network uses normalization. Batch normalization would make each critic score depend on the whole minibatch and couple the per-sample input gradients that the penalties measure.

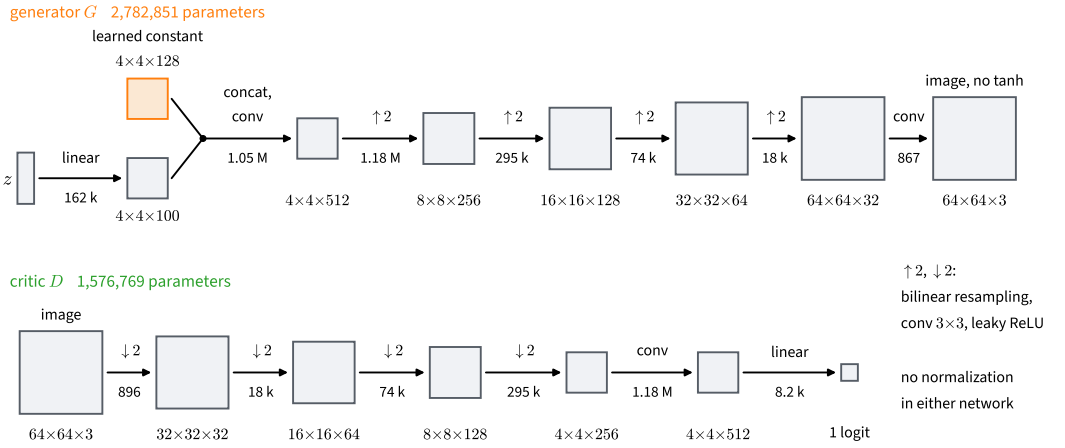


Figure 16.7: The minimal backbone used with the pairing loss, with the parameter count of each layer. The generator concatenates a learned $4 \times 4 \times 128$ constant with a linear projection of z to $4 \times 4 \times 100$, mixes them with a convolution, and doubles the resolution four times by bilinear upsampling, a 3×3 convolution and a leaky ReLU. A last convolution produces the image, with no tanh. The critic mirrors it with bilinear downsampling, and a linear head gives one logit. Neither network uses normalization.

The generator starts from a learned $4 \times 4 \times 128$ constant, which holds what all samples share. A linear layer projects z to a $4 \times 4 \times 100$ map, which is concatenated with the constant along the channel axis and mixed with it by a convolution. Both classes are saved: Section 16.6 reuses them at 32×32, which their `resolution` argument allows, and feeds the class label through the generator's `extra` input, concatenated beside the projected latent code.

Without normalization layers nothing rescales the activations, so the initialization has to keep their scale. We draw every weight with variance $2/((1 + 0.2^2)\, n_{\text{in}})$ for a layer with n_{in} inputs. This is the fan-in rule of He et al. (2015) with the gain of a leaky ReLU of slope 0.2, and it keeps the variance of the activations roughly constant from layer to layer.

```
def fan_in_init(module):
    """Fan-in normal weights with the leaky-ReLU gain, zero offsets."""
    if isinstance(module, (nn.Conv2d, nn.Linear)):
        nn.init.kaiming_normal_(module.weight, a=0.2)
        nn.init.zeros_(module.bias)


class Generator(nn.Module):
    """Learned 4x4 constant + projected latent (+ extra 4x4 input maps),
    bilinearly upsampled to resolution x resolution; no normalization."""
    def __init__(self, latent_dim, resolution, base_ch, *, extra_ch=0,
                 const_ch=128):
        super().__init__()
        self.latent_dim = latent_dim
        self.const = nn.Parameter(torch.randn(1, const_ch, 4, 4))
        self.z_proj = nn.Linear(latent_dim, 16 * latent_dim)
        self.mix = nn.Conv2d(const_ch + latent_dim + extra_ch, base_ch, 3,
                             padding=1)
        num_stages = int(np.log2(resolution / 4))       # 64: four doublings
        chans = [base_ch // 2 ** i for i in range(num_stages + 1)]
        self.stages = nn.ModuleList(
            [nn.Conv2d(c_in, c_out, 3, padding=1)
             for c_in, c_out in zip(chans[:-1], chans[1:])])
        self.to_rgb = nn.Conv2d(chans[-1], 3, 3, padding=1)
        self.act = nn.LeakyReLU(0.2)
        self.apply(fan_in_init)

    def forward(self, z, extra=None):
        zc = self.z_proj(z).reshape(-1, self.latent_dim, 4, 4)
        const = self.const.expand(z.shape[0], -1, -1, -1)
        maps = [const, zc] + ([] if extra is None else [extra])
        x = self.act(self.mix(torch.cat(maps, dim=1)))
        for conv in self.stages:                    # 4 -> 8 -> ... -> 64
            x = F.interpolate(x, scale_factor=2, mode='bilinear')
            x = self.act(conv(x))
        return self.to_rgb(x)                       # no tanh: raw output
```

The critic mirrors the generator. Four stages of convolution, leaky ReLU, and bilinear downsampling take a 64×64 image to 4×4, a mixing convolution gives the feature map `phi`, and a linear head returns one score.

```
class Discriminator(nn.Module):
    """Mirror of the generator: conv + leaky ReLU + bilinear downsampling to
    a 4x4 feature map phi(x), then a linear head on the flattened map."""
    def __init__(self, resolution, base_ch):
        super().__init__()
        num_stages = int(np.log2(resolution / 4))
        chans = [3] + [base_ch * 2 ** i for i in range(num_stages)]
        self.stages = nn.ModuleList(
            [nn.Conv2d(c_in, c_out, 3, padding=1)
             for c_in, c_out in zip(chans[:-1], chans[1:])])
        self.mix = nn.Conv2d(chans[-1], 2 * chans[-1], 3, padding=1)
```

```
        self.head = nn.Linear(2 * chans[-1] * 4 * 4, 1)
        self.act = nn.LeakyReLU(0.2)
        self.apply(fan_in_init)

    def phi(self, x):
        for conv in self.stages:                    # 64 -> 32 -> ... -> 4
            x = F.interpolate(self.act(conv(x)), scale_factor=0.5,
                              mode='bilinear', antialias=True)
        return self.act(self.mix(x))

    def forward(self, x):
        return self.head(self.phi(x).flatten(1))

for net in (Generator(latent_dim=100, resolution=64, base_ch=512),
            Discriminator(resolution=64, base_ch=32)):
    print(f'{type(net).__name__}: {d2l.num_params(net)} parameters')
```

```
Generator: 2782851 parameters
Discriminator: 1576769 parameters
```

Counted by `d2l.num_params` of Section 7.5, the generator has about 2.8 million parameters and the critic about 1.6 million, a reduced version of R3GAN's design without its residual blocks, grouped convolutions, and inverted bottlenecks.

16.5.2.3 Averaging, Flips, and Ranking

Three saved helpers complete the recipe. `sample_real` draws a training batch with random horizontal flips, which makes the real images harder for the critic to memorize. `EMA` keeps an exponential moving average of the generator's parameters, $\bar{\theta} \leftarrow \beta\bar{\theta} + (1-\beta)\theta$ after every step, here with a half-life of 500 steps, and we sample from the average. The iterates of the game rotate around the equilibrium as they contract (Section 16.4), so their average sits closer to it than they do. `pair_stats` reports two statistics of aligned (real, generated) pairs, read in the training traces below: the fraction that a critic orders correctly and the mean rank weight of Section 16.3.

```
class EMA:
    """Exponential moving average of a model's parameters, given as a
    half-life in steps. Buffers such as batch-norm statistics are copied;
    the average is kept in evaluation mode."""
    def __init__(self, model, half_life):
        import copy  # the d2l library preamble does not import it
        self.model = copy.deepcopy(model).requires_grad_(False).eval()
        self.decay = 0.5 ** (1 / half_life)

    def update(self, model):
        with torch.no_grad():
            for avg, p in zip(self.model.parameters(), model.parameters()):
                avg.lerp_(p, 1 - self.decay)
            for avg, b in zip(self.model.buffers(), model.buffers()):
                avg.copy_(b)
```

```
def sample_real(images, n, labels=None):
    """Draw a training batch with random horizontal flips; return it with
    its labels (None without labels)."""
    idx = torch.randint(0, len(images), (n,), device=images.device)
    batch = images[idx]
    flip = torch.rand(n, device=images.device) < 0.5
    batch = torch.where(flip.view(-1, 1, 1, 1), batch.flip(-1), batch)
    return batch, (None if labels is None else labels[idx])


def pair_stats(net_D, real, fake, y=None):
    """Ranking accuracy P(D(x) > D(x')) and mean rank weight
    E[sigma(D(x') - D(x))] over aligned (real, generated) pairs; given
    labels y, the critic is D(., y)."""
    critic = net_D if y is None else lambda x: net_D(x, y)
    with torch.no_grad():
        diff = critic(fake) - critic(real)
    return float((diff < 0).float().mean()), float(torch.sigmoid(diff).mean())
```

16.5.3 A Controlled Loss Comparison

16.5.3.1 Three Arms

We train the backbone three times from the same initialization, with Adam at $\beta_1 = 0$, $\beta_2 = 0.99$, and learning rate $2 \cdot 10^{-4}$, batch size 64, flips, and EMA; only the critic's objective changes. Writing x for a real image and x' for a generated one, each critic minimizes

$$L(D) + \frac{\gamma}{2}\Big(\mathbb{E}_x[\|\nabla_x D(x)\|^2] + \mathbb{E}_{x'}[\|\nabla_{x'} D(x')\|^2]\Big), \tag{16.35}$$

where L is either the logistic loss of Section 16.1, $\mathbb{E}_x[\log(1+e^{-D(x)})] + \mathbb{E}_{x'}[\log(1+e^{D(x')})]$ (`d2l.gan_loss_D`), or the pairing loss of Section 16.3, $\mathbb{E}_{x,x'}[\log(1+e^{D(x')-D(x)})]$ (`d2l.rpgan_loss_D`). The penalty terms R_1 and R_2 of Section 16.4 flatten the critic at the real and at the generated images, and each generator uses the non-saturating loss that matches its critic. At $q = p$ the best critic is constant, and the classic and pairing losses equal $2\log 2 \approx 1.39$ and $\log 2 \approx 0.69$.

arm	critic loss	penalty	steps
NS	classic	none	6,000
NS + R_1 + R_2	classic	$\gamma = 10$	12,000
RpGAN + R_1 + R_2	pairing	$\gamma = 10$	12,000

The first two arms isolate the penalty and the last two isolate pairing. Because γ is a squared length on the input scale (Section 16.4), a value tuned on one dataset does not transfer to another (Exercise 2). The unpenalized arm gets half the budget because it has failed by then.

The saved loop `train_r3gan` takes the two networks, the two losses as functions and the penalty weight as a number. It alternates `d2l.critic_step` and `d2l.generator_step` of Section 16.4 on flipped real batches and updates the EMA after every step. The critic step returns the loss and the batch means of R_1 and R_2, so the critic's steepness on real images, $\mathbb{E}_x\|\nabla_x D(x)\|^2$, is logged in every arm, at weight zero in the first. Every 250 steps the loop also applies `pair_stats` to 256 fresh real images and 256 fresh generated ones. It returns the EMA generator and the history, a dict of logged arrays. Given labels, it conditions each generated batch on the labels of its real batch, which Section 16.6 uses.

```
def train_r3gan(net_G, net_D, loss_D, loss_G, gamma, num_steps, images,
                labels=None, *, eval_images=None, eval_labels=None,
                eval_size=256, batch_size=64, lr=2e-4, half_life=500,
                log_every=250, seed=None):
    """Train with Adam (beta_1 = 0), alternating critic_step and
    generator_step on flipped real batches; a generated batch takes the
    labels of its real batch, if any. Every log_every steps, log the losses,
    R1, R2 and pair_stats on fresh pairs from eval_images (default: images).
    Return the EMA of the generator and the history, a dict of arrays."""
    if seed is not None:
        torch.manual_seed(seed)
    if eval_images is None:
        eval_images, eval_labels = images, labels
    opt_G, opt_D = [torch.optim.Adam(net.parameters(), lr, betas=(0.0, 0.99))
                    for net in (net_G, net_D)]
    noise = lambda n: torch.randn(n, net_G.latent_dim, device=images.device)
    generate = lambda z, y: net_G(z) if y is None else net_G(z, y)
    ema = EMA(net_G, half_life)
    history = {k: [] for k in ('step', 'loss_D', 'loss_G', 'R1', 'R2',
                               'accuracy', 'rank_weight')}
    for step in range(1, num_steps + 1):
        X, y = sample_real(images, batch_size, labels)
        with torch.no_grad():
            fake = generate(noise(batch_size), y)
        loss_d, r1, r2 = d2l.critic_step(net_D, opt_D, loss_D, X, fake,
                                         gamma, y)
        X, y = sample_real(images, batch_size, labels)
        loss_g = d2l.generator_step(net_G, opt_G, loss_G, net_D, X,
                                    noise(batch_size), y)
        ema.update(net_G)
        if step % log_every == 0:  # fresh pairs of real and generated images
            X, y = sample_real(eval_images, eval_size, eval_labels)
            with torch.no_grad():
                fake = generate(noise(eval_size), y)
            acc, weight = pair_stats(net_D, X, fake, y)
            logs = (step, loss_d, loss_g, r1, r2, acc, weight)
            for k, v in zip(history, logs):
                history[k].append(float(v))
    return ema.model, {k: np.array(v) for k, v in history.items()}
```

`train_arm` builds the two networks from the same seed in every arm, trains them, and returns the critic's train–holdout gap: the difference between its mean score on 1,024 training images and on 1,024 held-out images.

```
def train_arm(loss_D, loss_G, gamma, num_steps):
    torch.manual_seed(0)  # the same initialization in every arm
    net_G = Generator(100, 64, 512).to(device)
    net_D = Discriminator(64, 32).to(device)
    ema_G, history = train_r3gan(net_G, net_D, loss_D, loss_G, gamma,
                                 num_steps, train_imgs, seed=1)
    with torch.no_grad():
        gap = (net_D(train_imgs[:1024]).mean()
               - net_D(holdout_imgs[:1024]).mean())
    return ema_G, history, float(gap)
```

The three runs are the longest computation in the chapter, under half an hour on one GPU; a penalized step costs more because the penalty differentiates the critic's input gradient a second time. After each run we print the critic's loss and penalty, averaged over the last 2,000 steps, and the train–holdout gap.

```
arms = {'NS': (d2l.gan_loss_D, d2l.gan_loss_G, 0, 6000),
        'NS + R1 + R2': (d2l.gan_loss_D, d2l.gan_loss_G, 10, 12000),
        'RpGAN + R1 + R2': (d2l.rpgan_loss_D, d2l.rpgan_loss_G, 10, 12000)}
runs = {}
for name, (loss_D, loss_G, gamma, num_steps) in arms.items():
    ema_G, hist, gap = train_arm(loss_D, loss_G, gamma, num_steps)
    runs[name] = (ema_G, hist)
    loss, r1, r2 = (hist[k][-2000 // 250:].mean()  # the last 2,000 steps
                    for k in ('loss_D', 'R1', 'R2'))
    print(f'{name:16s} critic loss {loss:.2f}, '
          f'penalty {gamma / 2 * (r1 + r2):.3f}, train-holdout gap {gap:+.2f}')
```

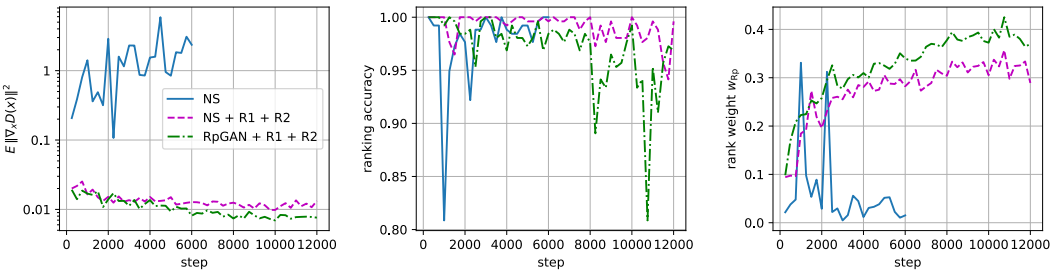
```
NS               critic loss 0.34, penalty 0.000, train-holdout gap +0.21
NS + R1 + R2     critic loss 1.12, penalty 0.121, train-holdout gap +0.03
RpGAN + R1 + R2  critic loss 0.50, penalty 0.085, train-holdout gap +0.01
```

Both penalized critics sit below their equilibrium values, about 1.05 against $2\log 2 \approx 1.39$ for the classic loss and about 0.5 against $\log 2 \approx 0.69$ for pairing: each stays somewhat ahead of its generator without running away. The penalty adds about a tenth to a fifth to the loss. The unpenalized critic's loss swings too widely to be read from one number. We return to the train–holdout gaps in Section 16.5.5.1.

16.5.3.2 Training Traces

The losses do not say how the critic separates the two sets, but three logged statistics do. The first is the critic's steepness on real images, $\mathbb{E}_x\|\nabla_x D(x)\|^2$, which R_1 pushes down. The second is the ranking accuracy $\hat{P}(D(x) > D(x'))$ on fresh pairs, where 0.5 means that the critic cannot tell the sets apart and 1 that it orders every pair. The third is the *rank weight* of Section 16.3, $w_{\mathrm{Rp}}(x') = \mathbb{E}_x[\sigma(D(x') - D(x))]$, averaged over generated samples: the probability, softened by the sigmoid, that a generated image outranks a real one. Replacing the sigmoid by a step function turns the averaged rank weight into one minus the ranking accuracy.

```
fig, axes = d2l.plt.subplots(1, 3, figsize=(12, 3.2))
hists = [hist for _, hist in runs.values()]
for ax, key, ylabel, yscale in zip(
        axes, ('R1', 'accuracy', 'rank_weight'),
        (r'$E\,\|\nabla_x D(x)\|^2$', 'ranking accuracy',
         r'rank weight $w_{\mathrm{Rp}}$'),
        ('log', 'linear', 'linear')):
    d2l.plot([h['step'] for h in hists], [h[key] for h in hists], 'step',
             ylabel, yscale=yscale, axes=ax)
axes[0].legend(list(runs), loc='center right')
fig.tight_layout()
```



The first panel separates the arms by two orders of magnitude. Without a penalty the critic's squared gradient on real images jumps between about 0.1 and a few, mostly near one, while both penalized critics stay flat near 0.01. The second panel hardly separates them: all three critics order more than nine in ten fresh pairs correctly at most checks, with occasional sharp dips. The penalty does not stop the critic from ranking; it limits how steep the critic is. In the third panel the unpenalized rank weight stays mostly below 0.2, often near zero, with spikes, while with the penalty it settles between about a quarter and two fifths.

16.5.3.3 Samples of the Three Arms

The grids show 64 samples of each EMA generator, drawn from the same latent codes in all three arms. The saved helper `tile_images` tiles a batch of images into one image, with pixel values clamped to $[-1, 1]$ and mapped to $[0, 1]$ for display.

```
def tile_images(imgs, num_rows):
    """Tile images (n, 3, h, w) in [-1, 1] into one (rows * h, cols * w, 3)
    array in [0, 1]."""
    n, c, h, w = imgs.shape
    grid = (imgs.clamp(-1, 1) + 1) / 2
    grid = grid.reshape(num_rows, n // num_rows, c, h, w)
    grid = grid.permute(0, 3, 1, 4, 2)  # (rows, h, cols, w, c)
    return grid.reshape(num_rows * h, -1, c).cpu().numpy()
```

```
torch.manual_seed(42)
z_show = torch.randn(64, 100, device=device)
with torch.no_grad():
    grids = [tile_images(ema_G(z_show), 8) for ema_G, _ in runs.values()]
d2l.show_images(grids, 1, 3, titles=list(runs), scale=4.5);
```



Without a penalty the classic loss collapses: after 6,000 steps the 64 latent codes produce one grey texture that resembles no sprite (in some runs a handful of images). The two penalized arms produce diverse, creature-shaped sprites on white backgrounds, with softer outlines than real sprites, and no difference between the two is visible.

The collapse is a training failure, not a minimum of the loss. Section 16.4 showed that gradient training circles the equilibrium unless something damps the critic near the data. Here nothing does: the unpenalized critic is about a hundred times steeper at the real images than the penalized ones. The rank weight shows what this does to the generator. Near zero, the critic ranks every generated sample below nearly every real one, and the weight no longer says which samples are worse, so the non-saturating generator, which weights a sample by $1 - w_{\mathrm{Rp}}$, gives every sample the same full push. With the penalty the weight stays graded, and the samples that the critic ranks lowest are pushed hardest. The ranking accuracy, the hard-threshold version of the same ordering, is close to one in all three arms and misses this difference.

With both penalties, the classic and the pairing loss reach the same quality on sprites at this budget: their scores below overlap across seeds, and so do their grids. Sprites have no countable modes, so this comparison cannot test coverage, the one place where the StackedMNIST ablation of Section 16.4 favored pairing.

16.5.4 Measuring Sample Quality

A grid shows a collapse, but it cannot rank two good checkpoints or two values of γ, and an implicit generator has no likelihood to rank them by. The standard scores pass n real and n generated images through a fixed feature map ϕ and compare the two clouds of feature vectors. Both scores used here are distances of Section 16.2 that have closed forms.

16.5.4.1 FID and KID from Two Closed Forms

The Fréchet inception distance (FID) of Heusel et al. (2017) fits a Gaussian to each cloud, $\mathcal{N}(\mu_p, \Sigma_p)$ to the real features and $\mathcal{N}(\mu_q, \Sigma_q)$ to the generated ones, and reports the squared Wasserstein-2 distance between the two Gaussians. In one dimension this is the cost of moving one bell curve onto the other, $(\mu_p - \mu_q)^2 + (\tau_p - \tau_q)^2$ for standard deviations τ_p and τ_q; real features $\mathcal{N}(0, 1)$ against generated features $\mathcal{N}(0.5, 1.1^2)$ give $0.25 + 0.01 = 0.26$. The general Wasserstein-2 problem has no explicit solution, but between Gaussians it does (Dowson and Landau 1982; Givens and Shortt 1984):

$$\mathrm{FID} = \left\|\mu_p - \mu_q\right\|^2 + \mathrm{tr}\Big(\Sigma_p + \Sigma_q - 2\big(\Sigma_p^{1/2}\,\Sigma_q\,\Sigma_p^{1/2}\big)^{1/2}\Big). \tag{16.36}$$

The first term compares the means and the trace term the spreads. If Σ_p and Σ_q share eigenvectors, with eigenvalues a_i and b_i, the trace term becomes $\sum_i (\sqrt{a_i} - \sqrt{b_i})^2$, the one-dimensional formula along each axis. FID sees only the first two moments of the features, and its plug-in estimate is biased at finite n, so two disjoint samples of real images score above zero.

The kernel inception distance (KID) of Bińkowski et al. (2018) uses the chapter's other closed form, the maximum mean discrepancy, with the cubic kernel $k(u, v) = (u^\top v/d + 1)^3$ on d-dimensional features. For n_p real images x_i and n_q generated images x'_j, KID reports the unbiased estimate of MMD^2 (`d2l.mmd2_unbiased`), which leaves out the terms that compare a point with itself:

$$\widehat{\mathrm{MMD}}^2 = \frac{1}{n_p(n_p-1)} \sum_{i \neq i'} k(\phi(x_i), \phi(x_{i'})) + \frac{1}{n_q(n_q-1)} \sum_{j \neq j'} k(\phi(x'_j), \phi(x'_{j'})) - \frac{2}{n_p n_q} \sum_{i,j} k(\phi(x_i), \phi($$

(16.37)

The first two sums measure how similar the images within each set are, the third how similar they are across sets. Because the estimate is unbiased, two samples of the same distribution give a value that fluctuates around zero and can be slightly negative.

```
def fid_score(feat_p, feat_q):
    """Frechet distance between Gaussians fitted to two feature sets."""
    feat_p = np.asarray(feat_p, np.float64)
    feat_q = np.asarray(feat_q, np.float64)
    mu_p, mu_q = feat_p.mean(0), feat_q.mean(0)
    C_p = np.cov(feat_p, rowvar=False)
    C_q = np.cov(feat_q, rowvar=False)
    lam, U = np.linalg.eigh(C_p)
    sqrt_C_p = (U * np.sqrt(np.maximum(lam, 0))) @ U.T
    M = sqrt_C_p @ C_q @ sqrt_C_p
    cross = np.linalg.eigvalsh((M + M.T) / 2)   # symmetric up to rounding
    return (((mu_p - mu_q) ** 2).sum() + np.trace(C_p) + np.trace(C_q)
            - 2 * np.sqrt(np.maximum(cross, 0)).sum())


def kid_score(feat_p, feat_q):
    """Unbiased MMD^2 between two feature sets, cubic polynomial kernel."""
    d = feat_p.shape[1]
    kernel = lambda a, b: (a @ b.T / d + 1) ** 3
    return d2l.mmd2_unbiased(kernel(feat_p, feat_p), kernel(feat_q, feat_q),
                             kernel(feat_p, feat_q))
```

16.5.4.2 A Feature Network

Published FID uses the pooled features of an Inception-v3 classifier trained on ImageNet. To keep the experiment inside the notebook, we train a small convolutional classifier on CIFAR-10 for 15 epochs and use its 128-dimensional penultimate layer as ϕ. A feature network can judge only images that resemble its training data. We therefore shrink the sprites to 32×32 by averaging blocks of 2×2 pixels, rather than enlarge CIFAR-10 to 64×64. Enlarged photographs are blurry, so a network trained on them has never seen sharp 64×64

outlines, and how it responds to them depends on its random initialization: in our runs, the FD of the same generated sprites varied by a factor of two to four between training runs of the network at 64×64, and by less than a factor of 1.5 at 32×32. The network's three blocks of 3×3 convolution and 2×2 pooling see windows of about 22 pixels, two thirds of the image side, so after global averaging its features describe parts of shapes, texture and color. The loader `d2l.load_cifar10_arrays`, the network `d2l.FeatureCNN`, and its training function `d2l.train_feature_cnn` (random flips, random shifts of up to four pixels, and a learning rate that rises and then anneals) contain nothing new; their code is in the online appendix "Utility Functions and Classes". Every evaluation pass below runs through `batched`, which applies a function to an array 1,000 rows at a time and concatenates the results.

```
def batched(f, X, batch_size=1000):
    """Apply f to X in batches, without gradients, and concatenate."""
    with torch.no_grad():
        return torch.cat([f(X_k) for X_k in X.split(batch_size)])
```

```
cifar_X, cifar_y, test_X, test_y = d2l.load_cifar10_arrays()
torch.manual_seed(0)
cnn = d2l.FeatureCNN(resolution=32).to(device)
d2l.train_feature_cnn(cnn, cifar_X, cifar_y, shift=4)
pred = batched(cnn, test_X).argmax(1)
print(f'CIFAR-10 test accuracy: {float((pred == test_y).float().mean()):.3f}')
```

```
CIFAR-10 test accuracy: 0.780
```

The classifier labels about four in five test images correctly.

16.5.4.3 Scoring the Runs

We score 500 samples of each generator against 500 held-out sprites. As a floor we also score two disjoint sets of 500 training images against each other: any distance between them is estimator noise and, for FD, finite-sample bias. Because the feature network is our own, we call the scores FD and MMD^2 rather than FID and KID; they use the same formulas but cannot be compared with published values. The backbone has no tanh, so its outputs can leave $[-1, 1]$. We clamp them before scoring, and the last column reports the fraction of raw pixel values outside the range.

```
features = lambda imgs: batched(  # 2 x 2 averages: 64 x 64 -> 32 x 32
    cnn.features, F.avg_pool2d(imgs, 2)).cpu().numpy()
n = 500
torch.manual_seed(7)
z_score = torch.randn(n, 100, device=device)
gens = {'DCGAN': net_G, **{name: ema_G for name, (ema_G, _) in runs.items()}}
with torch.no_grad():
    raw = {name: G(z_score) for name, G in gens.items()}
out_of_range = {name: float((x.abs() > 1).float().mean())
                for name, x in raw.items()}
feat_gen = {name: features(x.clamp(-1, 1)) for name, x in raw.items()}
feat_real = features(holdout_imgs[:n])
feat_a, feat_b = features(train_imgs[:n]), features(train_imgs[n:2 * n])
```

```
white = float((holdout_imgs == 1).float().mean())
```

Each feature set is compared with that of the held-out sprites; the first row is the floor.

```
print(f'{"run":18s}{"FD":>8s}{"MMD^2":>8s}{"out-of-range":>14s}')
print(f'{"real vs. real":18s}{fid_score(feat_a, feat_b):8.2f}'
      f'{kid_score(feat_a, feat_b):8.3f}')
for name, feat in feat_gen.items():
    print(f'{name:18s}{fid_score(feat, feat_real):8.2f}'
          f'{kid_score(feat, feat_real):8.3f}{out_of_range[name]:14.3f}')
print(f'real pixel values equal to +1: {white:.0%}')
```

```
run                     FD   MMD^2  out-of-range
real vs. real         1.29  -0.012
DCGAN                 3.58   0.198         0.000
NS                   69.15   7.852         0.006
NS + R1 + R2          4.56   0.598         0.141
RpGAN + R1 + R2       3.51   0.223         0.149
real pixel values equal to +1: 45%
```

The floor behaves as described above: FD is positive for two samples of the same distribution, while $\widehat{\mathrm{MMD}}^2$ is close to zero. Against it the collapsed arm scores dozens of times higher. The two penalized arms score three to five times the floor, which fits the soft outlines in their grids, and far below the collapsed arm. Which of the two scores lower changes from run to run, so this budget does not separate them. DCGAN lands close to them: the 2015 architecture keeps trainable the same classic loss that collapses on the unnormalized backbone. $\widehat{\mathrm{MMD}}^2$ agrees, with the collapsed arm far above the other three. Between about one in eight and one in six raw pixel values of a penalized generator lie outside $[-1, 1]$. Nearly half of the real pixel values equal $+1$, the white background, and this is where a generator without tanh overshoots; clamping maps the overshoot back to white.

Both scores depend on the feature network as well as on the generator: our classifier judges sprites by what helps to classify photographs, and a different feature network can change the values and reorder similar models.

16.5.5 Limited Data

16.5.5.1 Memorizing Critics

Forty thousand sprites is a small dataset for a critic with 1.6 million parameters: in 12,000 steps at batch size 64 it sees each training image about twenty times. Instead of estimating a density ratio, such a critic can start to recognize individual training images. It then scores them higher than held-out images, and its gradient pulls the generator toward the finite training set rather than the distribution behind it. The train–holdout gap printed after each run measures this. The penalized critics' gaps stay within a few hundredths of zero, so at this budget they score held-out sprites as they score their training images; the unpenalized critic's gap varies from run to run, by up to a few tenths in either direction, and lives on a larger score scale (Exercise 3). On smaller datasets this gap is the first sign of trouble (Karras, Aittala, et al. 2020). Whether the generator copies its training images is a separate question (Exercise 9).

16.5.5.2 Augmenting Both Inputs

Augmentation makes memorization harder, but where it is applied decides what the generator learns. Our loop flips only the real images, so the generator is trained toward the flipped distribution, which equals the data distribution only if sprites face left as often as right. Color jitter applied only to real images would likewise teach the generator to produce color shifts. Applying the same random transformation to both inputs of the critic, inside the generator's gradient path, avoids this (Zhao et al. 2020). The critic then sees many versions of each training image, and the matching problem stays the original one as long as the transformation cannot map two different distributions to the same one. Karras, Aittala, et al. (2020) call such a transformation *non-leaking* (Exercise 10).

We can now build an image generator from either decade, change its loss one ingredient at a time, and score the result against a floor whose meaning we know. The generator still answers no request: it returns a random sprite, never the one we ask for. Section 16.6 gives both players the request.

16.5.6 Summary

DCGAN trained the classic loss to diverse sprites through its architecture: transposed convolutions, batch normalization, and Adam with $\beta_1 = 0.5$. A backbone without normalization relies on the loss instead. On it, the classic loss without a penalty collapses within 6,000 steps, while with the zero-centered penalties R_1 and R_2 both the classic and the pairing loss train to diverse sprites and tie. The penalty does not stop the critic from ranking; it makes the critic about a hundred times less steep, so the rank weight stays graded instead of saturating near zero. FID and KID are the Gaussian Wasserstein-2 distance and an unbiased cubic-kernel MMD in a feature space; their values depend on the feature network and on n, so we read them against a real-versus-real floor. On limited data the critic memorizes first, and augmenting both of its inputs counters this.

16.5.7 Exercises

1. **Checkerboards from stride.** `G_block` upsamples with a transposed convolution of kernel 4, stride 2, and padding 1.
    1. Find every kernel size (at most 6) and padding for which a transposed convolution with stride 2 exactly doubles the side n.
    2. A one-dimensional transposed convolution with stride 2 and a kernel of ones acts on a constant input. Away from the borders, how many input positions contribute to each output position for kernel size 3, and for kernel size 4? The first case is what Odena et al. (2016) call uneven overlap.
    3. `Generator` instead upsamples bilinearly and then applies a 3×3 convolution with stride 1. What does this produce from a constant input away from the borders, and why can no choice of weights create a periodic pattern?
2. **Penalty weight and resolution.** R3GAN tunes γ for every dataset and resolution. Part of this follows from the units of the input gradient.
    1. Let U upsample a 64×64 image to 128×128 by copying each pixel into a 2×2 block, and let A average each block, so that the critic $D' = D \circ A$ scores Ux as D scores x. Express $\|\nabla D'(Ux)\|^2$ through $\|\nabla D(x)\|^2$. Which γ' makes the two penalized games equivalent?

2. How must γ change if the pixel values lie in $[0, 1]$ instead of $[-1, 1]$?
3. Karras, Aittala, et al. (2020) set $\gamma = 0.0002\, r^2/B$ for resolution r and batch size B. Which factor does the upsampling argument explain?

3. code **A scale-free overfitting statistic.** The printed train–holdout gaps of the three arms live on different score scales. Karras, Aittala, et al. (2020) consider

$$r_v = \frac{\mathbb{E}[D_{\text{train}}] - \mathbb{E}[D_{\text{holdout}}]}{\mathbb{E}[D_{\text{train}}] - \mathbb{E}[D_{\text{generated}}]}, \qquad r_t = \mathbb{E}\big[\,\text{sign}(D_{\text{train}})\big],$$

where the subscripts name the images the critic scores.

1. Show that r_v is unchanged when D is replaced by $aD + c$ with $a > 0$, whereas the gap scales with a and r_t changes with c.
2. Adding a constant to a pairing critic leaves its loss unchanged, so the sign of one score means nothing. Under which transformations of D is $\mathbb{E}[\text{sign}(D_{\text{train}} - D_{\text{holdout}})]$ invariant, and what is its value for a critic that scores training and held-out images alike?
3. Compute both statistics for the three arms from 1,024 images of each kind. Do they change the comparison of the arms?

4. **FID across dimensions.** In one dimension the squared Wasserstein-2 distance is $\int_0^1 \big(Q_p(u) - Q_q(u)\big)^2\, du$ for quantile functions Q_p and Q_q.

1. Using $Q_p(u) = \mu_p + \tau_p\, Q_0(u)$, with Q_0 the quantile function of $\mathcal{N}(0, 1)$, show that two univariate Gaussians are at distance $(\mu_p - \mu_q)^2 + (\tau_p - \tau_q)^2$.
2. Evaluate Equation 16.36 for $\mathcal{N}(0, I_d)$ against $\mathcal{N}(0, c\, I_d)$ and against $\mathcal{N}(\mu_0 \mathbf{1}, I_d)$, where $\mathbf{1}$ is the all-ones vector.
3. A generator's features are 10% too spread out in every coordinate, $c = 1.21$. Compare its FID in 128 dimensions, as here, and in the 2,048 dimensions of Inception. What does this imply for scores from different feature networks?

5. code **Finite-sample bias.**

1. Compute the real-versus-real FD and $\widehat{\text{MMD}}^2$ between two disjoint sets of n held-out sprites for $n \in \{100, 250, 500, 1000, 2000\}$, five random splits each. How does each move with n?
2. Fit a line to FD against $1/n$ and extrapolate to $1/n \to 0$, the bias correction of Chong and Forsyth (2020) . How close to zero is the result, and how large is the bias at $n = 500$?
3. Repeat the fit for the RpGAN + R_1 + R_2 generator. Is the slope the same? Why can two generators rank differently at $n = 500$ and as $n \to \infty$?

6. code **Precision and recall.** Kynkäänniemi et al. (2019) estimate the support of a feature set by balls around its points, with radii the distances to the K-th nearest neighbors. Precision is the fraction of generated features inside the real support, recall the fraction of real features inside the generated support.

1. Implement both with $K = 3$ for `feat_real` against 500 samples of each generator and against 500 training images. How do the metrics respond to $K \in \{1, 10\}$?
2. A generator collapses onto J distinct images, each produced equally often by the 500 samples. For $K = 3$, find the largest J for which every radius is zero, and the

recall of such a generator. What does this say about recall as a count of covered modes?

7. code **Initialization and collapse.** ▣ Let the diversity of a generator be the mean pairwise feature distance of its samples from `z_show`, divided by the same statistic for 64 held-out images. Keep every run to 5,000 steps.
   1. Record the diversity of the raw generator every 500 steps. Train the unpenalized arm from the fan-in initialization and with all weights drawn from a normal distribution with standard deviation 0.01, 0.02 (the DCGAN convention), or 0.05. Does any initialization keep the diversity above one half? Does the rise of $\mathbb{E}\|\nabla_x D(x)\|^2$ precede its fall?
   2. Repeat the fan-in run with three seeds. How much does the step at which the diversity falls below one half vary, and what does this imply for a loss comparison based on a single run?
8. **Moments behind FID and KID.** ▣ FID sees only means and covariances.
   1. Expand the KID kernel $(u^\top v/d+1)^3$ with the binomial theorem and $(u^\top v)^j = \langle u^{\otimes j}, v^{\otimes j}\rangle$ to show that the population MMD^2 is
   $$\sum_{j=1}^{3} \binom{3}{j} d^{-j} \left\| \mathbb{E}_p[u^{\otimes j}] - \mathbb{E}_q[u^{\otimes j}] \right\|^2. \tag{16.38}$$
   Which moments does KID compare?
   2. In one dimension, compute the population FID and MMD^2 of $p = \mathcal{N}(0,1)$ against $X-1$ for X exponential with rate 1 (mean 0, variance 1, third moment 2), and against the equal mixture of $\mathcal{N}(\pm a, 1-a^2)$ for $0<a<1$.
   3. Is the KID kernel *characteristic*, that is, does MMD$^2 = 0$ imply $q = p$? Does a Gaussian kernel distinguish the mixture in (b) from p?
9. [short-code] **Checking for copies.** Embed the 500 samples of the RpGAN + R_1 + R_2 arm drawn from `z_score` and all training images with `features`, and find each sample's nearest training image.
   1. Compare the median nearest-neighbor distance of the samples with that of the held-out sprites in `feat_real`; a copying generator would sit closer to its training images than new sprites do. Variants of one species fall on both sides of the split. Does this make the test more or less likely to flag copying?
   2. Show the eight samples closest to the training set above their nearest training images. Do the pairs look alike?
   3. The features describe texture and color in windows of about 22 pixels. Could the test detect a generator that recombines memorized parts of different training images?
10. **Leaking augmentations.** ▣ Let F flip an image horizontally and $F_{\#}r$ be the distribution of flipped images under r. Flipping with probability ρ maps r to $T_\rho r = (1-\rho)\, r + \rho\, F_{\#}r$.
    1. `sample_real` flips only real images, with $\rho = 1/2$, so the generator is trained toward $T_{1/2}\, p$. If a fraction α of the sprites faces left with no mirrored counterpart

and every other sprite is its own mirror image, which fraction of the optimal generator's samples faces right?

2. A differentiable augmentation applies T_ρ to real and generated images. Using $F_\# F_\# r = r$, show that $T_\rho q = T_\rho p$ implies $q = p$ for $\rho \neq 1/2$. For $\rho = 1/2$, which generator distributions can the critic not tell from p? Give one that never produces an unmirrored sprite.
3. With probability ρ, an augmentation rotates each image by a multiple of 90° chosen uniformly. Show that it is non-leaking for $\rho < 1$ and leaking at $\rho = 1$. Why can a finite critic on finite data fail well before $\rho = 1$, as Karras, Aittala, et al. (2020) observe above about 0.85?

16.6 Conditional Generation

The garment generator of Section 16.1 and the sprite generators of Section 16.5 take no request. Each returns a sample from the whole data distribution: to get a pair of trousers from the garment generator we can only draw again until one appears, and nothing lets us ask the sprite generator for a particular creature. Most image generators that people use take a request instead: a class, a caption, a sketch to complete. Within months of the original GAN, Mirza and Osindero (2014) proposed the obvious repair: give the request to both players. The generator receives it as an extra input, and the critic judges whether a sample is a realistic instance of that request.

Is that one change enough? This section asks what the game then optimizes, how the critic should read the request, and how we can tell whether the generator obeys. The answers reuse the chapter's log-loss game, pairing loss and penalties, applied once per request. The running example is CIFAR-10: ask for a bird, and check that we get one. The check turns out to be the harder half, because a generator can obey every request and still return a single image per class.

16.6.1 The Conditional Game

16.6.1.1 The Optimal Conditional Critic

A condition specifies a property of the requested sample. It may be a class label, a caption embedding, or an entire image; our running example uses the ten CIFAR-10 class labels. The data consists of image–label pairs drawn from a joint distribution $p(x, c)$. The generator also receives the condition, so $x' = G(z, c)$, and varying z at fixed c samples from a conditional distribution $q(\cdot \mid c)$. We draw the generator's conditions from the data marginal $p(c)$, so the generated pairs follow $q(x, c) = p(c)\, q(x \mid c)$. The real and generated joint distributions then share the condition marginal and differ only in their conditional image distributions.

The critic scores sample–condition pairs, $D(x, c)$, and the game is the log-loss game of Section 16.1 played on those pairs:

$$V(D) \;=\; \mathbb{E}_{(x,c)\sim p}\big[\log \sigma(D(x,c))\big] + \mathbb{E}_{c\sim p(c),\; x'\sim q(\cdot|c)}\big[\log \sigma(-D(x',c))\big]. \tag{16.39}$$

In the unconditional game the optimal critic is the log density ratio $\log(p/q)$, and the value at that critic is $2\,\mathrm{JS}(p, q) - 2\log 2$. The pointwise argument behind both results works for

any input space, so it applies with the pair (x, c) in place of x. The optimal critic is the log ratio of the two joint densities, and their shared condition marginal cancels:

$$D^\star(x,c) \;=\; \log \frac{p(x,c)}{q(x,c)} \;=\; \log \frac{p(c)\,p(x \mid c)}{p(c)\,q(x \mid c)} \;=\; \log \frac{p(x \mid c)}{q(x \mid c)}. \tag{16.40}$$

The optimal conditional critic is the log density ratio within the slice of data that the condition selects. For the bird condition it compares real and generated birds; images of other classes do not enter. Substituting this critic into Equation 16.39 keeps the outer expectation over c, and each condition contributes the value of its own unconditional game:

$$\max_D V(D) \;=\; \mathbb{E}_c\Big[\,2\,\mathrm{JS}(p(\cdot \mid c),\, q(\cdot \mid c))\Big] - 2\log 2. \tag{16.41}$$

Thus the conditional game is an average of per-condition games, weighted by how often each condition occurs. Its value reaches the minimum $-2\log 2$ exactly when $q(\cdot \mid c) = p(\cdot \mid c)$ for every condition that $p(c)$ weights: the generator must match the data within every slice.

16.6.1.2 Stronger than Marginal Matching

This requirement is stronger than matching the distribution of images. Consider a generator that produces accurate CIFAR-10 images but shifts every label cyclically, so that $q(\cdot \mid c) = p(\cdot \mid c{+}1)$: a request for a bird returns a cat. All ten classes occur equally often, so the shift leaves the mixture over classes unchanged. The generator's distribution of images equals the data's, and the unconditional game sits at its minimum, $-2\log 2$; it cannot see the wrong labels. The conditional game compares each slice with its neighbor instead. If the classes had disjoint supports, every slice would contribute $\mathrm{JS} = \log 2$, and Equation 16.41 would equal 0, the largest value the game can take. CIFAR-10 classes overlap a little, so the value sits just below that maximum: a perfect image generator that fails every request is close to the worst conditional generator (Exercise 1 computes both values for two Gaussian slices).

The weighting by $p(c)$ has a cost for imbalanced data. A rare condition contributes little to the average, so the generator can serve it poorly at a small cost to the objective (Exercise 6).

16.6.1.3 Same-Class Pairs

Our experiment trains with the pairing loss of Section 16.3, which compares a real sample with a generated one. We pair each real image with a generated image of the same class, a real bird with a generated bird. Within one class this is the pairing game between $p(\cdot \mid c)$ and $q(\cdot \mid c)$, with the objective $\Phi_c(D) = \mathbb{E}_{x\sim p(\cdot|c),\, x'\sim q(\cdot|c)}[\log \sigma(D(x,c) - D(x',c))]$. For one pair of distributions, Section 16.3 showed that the value of the pairing game is the Jensen–Shannon divergence between the two orderings of a pair, real sample first ($p \otimes q$) or generated sample first ($q \otimes p$). The critic of each class can be optimized separately, so the conditional value averages these divergences over classes:

$$\max_D \mathbb{E}_c\big[\Phi_c(D)\big] + \log 2 \;=\; \mathbb{E}_c\Big[\mathrm{JS}\big(p(\cdot \mid c) \otimes q(\cdot \mid c),\; q(\cdot \mid c) \otimes p(\cdot \mid c)\big)\Big]. \tag{16.42}$$

Each term is zero exactly when its slice is matched, so the fixed point is that of Equation 16.41. The shift invariance of the pairing game carries over class by class. Adding a constant to the critic leaves every score difference unchanged, and with same-class pairs this holds for one constant per class: $D(x, c) + \zeta_c$ has the same loss for any constants $\zeta_1, \dots, \zeta_{10}$. The critic never calibrates its scores across classes, so our diagnostics must not depend on these constants (Exercise 3 measures them). The chapter's other results, such as the analysis of the zero-centered penalties, likewise hold class by class, since each concerns one pair of distributions.

16.6.2 Conditioning Mechanisms

16.6.2.1 Three Ways In

The analysis says what the optimal critic computes, but not how a network should take the condition in. Three mechanisms are common (Figure 16.8).

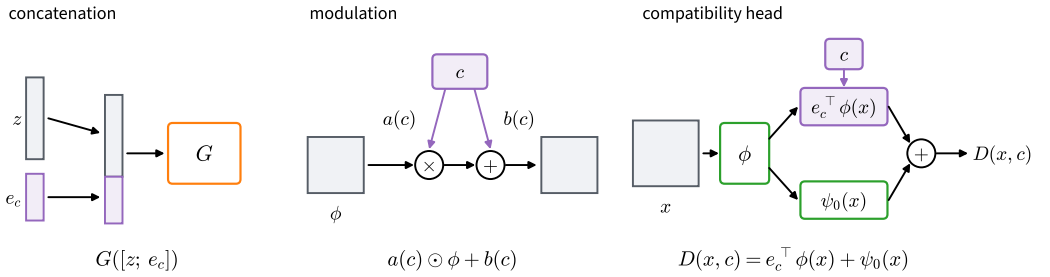


Figure 16.8: Three ways a condition c enters a GAN. Left: concatenation appends a class embedding e_c to the latent code (or to an image or feature map in the critic). Middle: modulation scales and shifts a feature map ϕ by functions of the condition, $a(c) \odot \phi + b(c)$ (conditional batch normalization normalizes ϕ first). Right: the compatibility head of the projection critic scores the inner product of a class embedding with the critic features, $e_c^\top \phi(x)$, and adds an unconditional term $\psi_0(x)$.

Concatenation (Mirza and Osindero 2014) embeds the condition in a vector e_c and appends it to an input: beside the latent code in the generator, or beside the image or a feature map in the critic. It imposes no form on how sample and condition interact, so the network must learn that interaction from the adversarial signal alone. *Modulation* scales and shifts a feature map, $a(c) \odot \phi + b(c)$, with one entry per channel; BigGAN computes $a(c)$ and $b(c)$ from a class embedding in every block of its generator (Brock et al. 2019). Papers write $\gamma(c)$ and $\beta(c)$, but γ is the penalty weight in this chapter. The third mechanism conditions only the critic, through a head that scores how well sample and condition fit together. Its most common form follows from the optimal critic.

16.6.2.2 The Projection Head from Bayes' Rule

Bayes' rule, $p(x \mid c) = p(c \mid x)\, p(x)/p(c)$, applied to the numerator and the denominator of Equation 16.40, cancels the shared label marginal and yields

$$\log \frac{p(x \mid c)}{q(x \mid c)} \;=\; \log \frac{p(c \mid x)}{q(c \mid x)} + \log \frac{p(x)}{q(x)}. \tag{16.43}$$

Thus the optimal conditional critic separates condition compatibility from unconditional sample quality. The second term is the log density ratio of the unconditional game. The first compares the real and generated *label posteriors* at x. A generator that produces realistic images with scrambled labels differs from the data through the first term; one that produces poorly formed images of the requested class differs through the second.

If both posteriors are softmax classifiers on one shared feature map ϕ, the first term takes a familiar form. Let $p(c \mid x) \propto \exp(u_c^\top \phi(x))$ with one weight vector per class, and $q(c \mid x) \propto \exp(w_c^\top \phi(x))$ likewise. The two normalizers do not depend on c, so the log ratio of the posteriors is the inner product $(u_c - w_c)^\top \phi(x)$ plus a function of x alone. Absorbing that function, together with the unconditional term, into a single head ψ_0 and writing $e_c = u_c - w_c$, the critic becomes

$$D(x, c) \;=\; e_c^\top \phi(x) + \psi_0(x). \tag{16.44}$$

This is the *projection discriminator* of Miyato and Koyama (2018): one learned embedding per class, an inner product with the critic's features, and an unconditional head. Bayes' rule is an identity, but the shared feature map is a modeling assumption (Exercise 2). A linear ψ_0 can be folded into the class embeddings, and R3GAN's conditional critic keeps only the inner product (Huang et al. 2024); StyleGAN-T replaces the class embedding by the CLIP embedding of a caption, which makes the same head the text-conditioned critic of current one-step generators (Sauer, Karras, et al. 2023). The *auxiliary classifier* critic of Odena et al. (2017) is the main alternative: it adds a classification head, and the generator is trained to make its samples receive the requested label, which favors easily classified prototypes over within-class diversity (Exercise 7).

Our experiment uses concatenation in the generator and a projection head in the critic, the division R3GAN uses in its class-conditional experiments. The critic gets the structure that the derivation motivates; no such identity says how the generator should use c, so concatenation leaves that free.

16.6.2.3 An Image as the Condition

The condition can also be an entire image: a semantic map, an edge sketch, or a grayscale photograph. Pix2pix (Isola et al. 2017) plays the conditional game on output–input pairs, with the source image as c, and adds an L_1 loss against the paired target. The source fixes the coarse layout of the output, which the L_1 term recovers. Fine texture is ambiguous given the source, and the L_1 term alone blurs it, so pix2pix keeps the critic beside it.

16.6.3 Class-Conditional CIFAR-10

We now train a class-conditional generator on the 50,000 labeled 32×32 training images of CIFAR-10, which come from ten classes and are loaded with `d2l.load_cifar10_arrays` (the online appendix "Utility Functions and Classes"). R3GAN reports a class-conditional CIFAR-10 benchmark trained for four days on eight GPUs. Our run takes 15,000 steps, well under half an hour on one GPU, and tests the conditioning and its measurement, not large-scale sample quality.

```
classes = ['airplane', 'automobile', 'bird', 'cat', 'deer',
           'dog', 'frog', 'horse', 'ship', 'truck']
```

```
device = d2l.try_gpu()
train_X, train_y, test_X, test_y = d2l.load_cifar10_arrays()
first = [int((train_y == c).nonzero()[0]) for c in range(10)]
d2l.show_images(train_X[first].cpu().permute(0, 2, 3, 1) / 2 + 0.5,
                num_rows=2, num_cols=5, titles=classes);
```



16.6.3.1 A Conditional Backbone

The networks are the saved `d2l.Generator` and `d2l.Discriminator` of Section 16.5 at resolution 32, one resampling stage fewer than for the 64×64 sprites, with the condition as an extra input to each. Everything else is unchanged: bilinear resampling, leaky ReLU, no normalization, no tanh, and fan-in initialization. `CondGenerator` embeds the class, broadcasts the embedding over the 4×4 grid, and passes it as the base class's `extra` input, which is concatenated with the learned constant and the projected latent code before the mixing convolution.

```
class CondGenerator(d2l.Generator):
    """The backbone generator with a class embedding as extra 4x4 maps."""
    def __init__(self, latent_dim, resolution, base_ch, embed_dim=64,
                 num_classes=10):
        super().__init__(latent_dim, resolution, base_ch, extra_ch=embed_dim)
        self.embed = nn.Embedding(num_classes, embed_dim)  # unit variance

    def forward(self, z, y):
        ec = self.embed(y)[:, :, None, None].expand(-1, -1, 4, 4)
        return super().forward(z, ec)
```

`ProjectionCritic` implements Equation 16.44. The base class's `phi` returns the 4×4 feature map after the mixing convolution; summing it over positions gives $\phi(x)$. A linear head on $\phi(x)$ gives $\psi_0(x)$ and replaces the base head, which reads the flattened map, and a table holds one embedding e_c per class. The table starts at zero, so the critic begins as an unconditional critic and learns the label term from the data.

```
class ProjectionCritic(d2l.Discriminator):
    """The backbone critic with a projection head: e_c . phi(x) + psi_0(x)."""
    def __init__(self, resolution, base_ch, num_classes=10):
        super().__init__(resolution, base_ch)
        width = self.mix.out_channels
        self.head = nn.Linear(width, 1)  # psi_0 on the pooled features
        d2l.fan_in_init(self.head)
        self.embed = nn.Embedding(num_classes, width)
        nn.init.zeros_(self.embed.weight)  # start as an unconditional critic

    def forward(self, x, y):
        phi = self.phi(x).sum(dim=(2, 3))  # sum over the 4x4 positions
        return (self.embed(y) * phi).sum(dim=1) + self.head(phi).squeeze(-1)

net_G, net_D = CondGenerator(100, 32, 256), ProjectionCritic(32, 64)
print(net_G(torch.zeros(2, 100), torch.zeros(2, dtype=torch.long)).shape)
print(net_D(torch.zeros(2, 3, 32, 32),
            torch.zeros(2, dtype=torch.long)).shape)
```

```
torch.Size([2, 3, 32, 32])
torch.Size([2])
```

16.6.3.2 Training with Same-Class Pairs

Training follows the sprite recipe of Section 16.5 and runs its saved loop `d2l.train_r3gan`: the pairing loss for both players, the penalties R_1 and R_2 on real and generated images, Adam with $\beta_1 = 0$, random horizontal flips of the real images, and an exponential moving average of the generator (`d2l.EMA`, half-life 500 steps) that supplies the generator for evaluation. Same-class pairs come from the labels: given them, the loop draws a labeled real batch and conditions the generated batch on the same labels. With the labels as `y`, the saved steps `d2l.critic_step` and `d2l.generator_step` of Section 16.4 score with the critic $x \mapsto D(x, y)$, so the losses and the penalty see an ordinary critic of images.

With the test images as its evaluation images, the ranking accuracy that the loop logs every 250 steps becomes the *held-out pair accuracy*: on 512 fresh pairs of a test image and a generated image of the same class, the fraction that the critic orders correctly, $\hat{P}(D(x, c) > D(x', c))$. The constants ζ_c cancel from it, whereas the critic's mean score drifts with them. A value of one half means that the critic cannot tell real from generated images of a class.

The runs use the penalty weight $\gamma = 0.5$, against $\gamma = 10$ for the sprites. The penalty weight is a squared length on the scale of the input (Section 16.4), and the right value also depends on the critic, the batch size and the budget, so a new setup needs its own sweep (Exercise 8).

```
torch.manual_seed(0)  # the initialization
net_G = CondGenerator(100, 32, 256).to(device)
net_D = ProjectionCritic(32, 64).to(device)
gamma = 0.5
ema_G, history = d2l.train_r3gan(
    net_G, net_D, d2l.rpgan_loss_D, d2l.rpgan_loss_G, gamma, 15000, train_X,
    train_y, eval_images=test_X, eval_labels=test_y, eval_size=512,
```

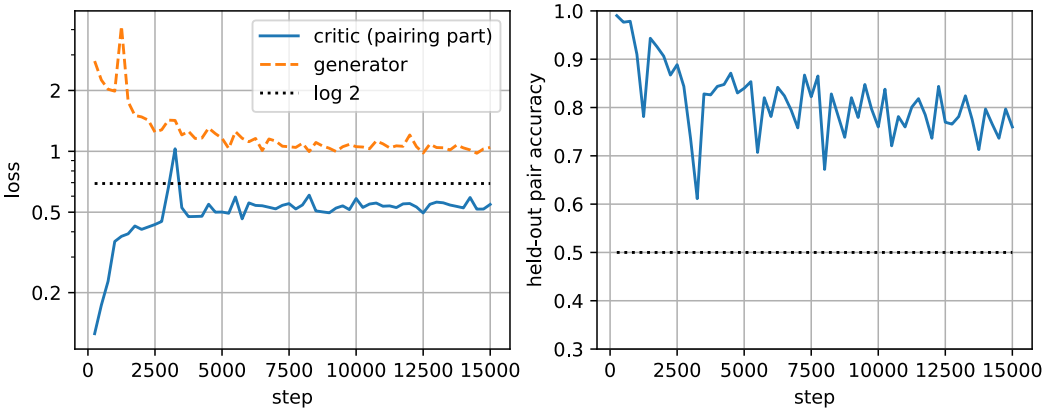
```
    batch_size=128, seed=1)
```

We average the logs over the last 5,000 steps.

```
loss_D, loss_G, pen, acc = (np.mean(v[-5000 // 250:]) for v in (
    history['loss_D'], history['loss_G'], history['R1'] + history['R2'],
    history['accuracy']))
print(f'critic loss {loss_D:.2f} (+ penalty {gamma / 2 * pen:.3f}), '
      f'generator loss {loss_G:.2f}, pair accuracy {acc:.2f}')
```

```
critic loss 0.54 (+ penalty 0.060), generator loss 1.05, pair accuracy 0.78
```

```
steps = history['step']
fig, axes = d2l.plt.subplots(1, 2, figsize=(9, 3.2))
d2l.plot(steps, [history['loss_D'], history['loss_G'],
                 np.full_like(steps, np.log(2))], 'step', 'loss',
         legend=['critic (pairing part)', 'generator', 'log 2'],
         fmts=('-', '--', 'k:'), yscale='log', axes=axes[0])
d2l.plot(steps, [history['accuracy'], np.full_like(steps, 0.5)], 'step',
         'held-out pair accuracy', ylim=[0.3, 1], fmts=('-', 'k:'),
         axes=axes[1]);
```



The dotted lines mark log 2, the value of either pairing loss when the critic cannot order a pair, and a pair accuracy of one half. Both losses settle within the first few thousand steps, after a high start of the generator loss: the generator's near 1.0, above log 2, and the critic's near 0.5, below it, with the penalty adding about 0.06. The critic still wins most comparisons: late in training it orders about 80% of fresh same-class pairs correctly, down from about 90% early in training. At this budget the generator has not caught up with the data.

16.6.3.3 Samples on Request

Each panel of the next figure requests one class, and its nine images use fresh latent codes; `d2l.tile_images` of Section 16.5 tiles them. As in that section, the generator has no tanh, so the display and every evaluation below clamp its output to $[-1, 1]$.

```
torch.manual_seed(42)
z = torch.randn(90, 100, device=device)
y = torch.arange(10, device=device).repeat_interleave(9)  # nine per class
with torch.no_grad():
    imgs = ema_G(z, y)
panels = [d2l.tile_images(imgs[9 * c:9 * (c + 1)], 3) for c in range(10)]
d2l.show_images(panels, 2, 5, titles=classes, scale=2);
```



Most panels read as their class. Vehicles are recognizable by shape: cars and trucks as wheeled bodies, ships as hulls on water, airplanes against the sky. The animal panels lean more on palette and background: frogs are green and brown textures, deer and horses brown forms on grass, while birds, cats and dogs are often similar brown shapes. Pose, color and background vary within every panel, so the figure shows no obvious collapse within a class. Inspection cannot say how often a generated image would be recognized as its class, or how much variety is missing; we measure both next.

16.6.4 Measuring Condition Alignment

16.6.4.1 A Referee Classifier

An unconditional metric cannot detect the cyclic label shift, which preserves the distribution of images while violating every request. We therefore check requests with a CIFAR-10 classifier, the *referee*. It is the feature network of Section 16.5, trained the same way: `d2l.train_feature_cnn` at CIFAR-10's native 32×32 resolution with random horizontal flips, random shifts of up to four pixels, and a learning rate that rises and then anneals over 15 epochs. Without the augmentation and the schedule the network overfits, and its accuracy on a single class moves by tens of points between two training seeds, more than the differences we want to read. With them it reaches about 78%, and its per-class accuracies move by a few points.

```
torch.manual_seed(0)
cnn = d2l.FeatureCNN(resolution=32).to(device)
d2l.train_feature_cnn(cnn, train_X, train_y, shift=4)
pred = d2l.batched(cnn, test_X).argmax(1)
```

```
cls_acc = [float((pred[test_y == c] == c).float().mean())
           for c in range(10)]
print(f'CIFAR-10 test accuracy: {float((pred == test_y).float().mean()):.3f}')
```

```
CIFAR-10 test accuracy: 0.779
```

16.6.4.2 Alignment per Class

For each generated image, the referee either predicts the requested class or it does not. The fraction of matches, overall and per class, is the *condition alignment*. We compare it with the referee's accuracy on real test images of the same class, because the referee also misclassifies real images. The real accuracy is a reference, not an upper bound: unusually prototypical generated images can be easier to classify than photographs. The next cell generates 500 images of each class and the one after it prints both and their difference, the *gap*.

```
torch.manual_seed(7)
y_req = torch.arange(10, device=device).repeat_interleave(500)  # requests
with torch.no_grad():
    pred_req = cnn(ema_G(torch.randn(5000, 100, device=device), y_req)
                   .clamp(-1, 1)).argmax(1)
align = [float((pred_req[y_req == c] == c).float().mean()) for c in range(10)]
```

```
print(f'{"class":12s}{"alignment":>11s}{"real":>7s}{"gap":>7s}')
for name, a, r in zip(classes + ['mean'], align + [np.mean(align)],
                      cls_acc + [np.mean(cls_acc)]):
    print(f'{name:12s}{a:11.2f}{r:7.2f}{a - r:+7.2f}')
```

```
class         alignment   real    gap
airplane           0.70   0.78  -0.08
automobile         0.84   0.91  -0.07
bird               0.67   0.66  +0.01
cat                0.50   0.58  -0.08
deer               0.69   0.71  -0.02
dog                0.69   0.76  -0.07
frog               0.80   0.85  -0.06
horse              0.71   0.80  -0.09
ship               0.85   0.87  -0.02
truck              0.82   0.87  -0.05
mean               0.73   0.78  -0.05
```

Averaged over classes, alignment is about 0.7, six or seven points below the referee's accuracy on real images. No class is recognized much more often than its real counterpart, and no class is harder than cat in either column. Individual gaps move by several points between runs, so the table does not rank them.

16.6.4.3 Fidelity and Within-Class Diversity

Alignment does not measure image quality. We therefore also compute the two feature-distribution distances of Section 16.5: the Fréchet distance between Gaussian fits to two feature sets, and the unbiased MMD^2 estimate with a cubic polynomial kernel. Both use

the referee's 128-dimensional features, so we call them FD and MMD^2 (CIFAR-CNN); they cannot be compared with published FID or KID scores. The evaluation compares 1,000 generated images, 100 per class, with 1,000 test images in the same proportions, and a second disjoint set of 1,000 test images gives the real-versus-real floor. The distances ignore the labels, so the table is followed by the *within-class feature variance*: the variance of the features within each class, averaged over features and classes.

```
features = lambda imgs: d2l.batched(cnn.features, imgs).cpu().numpy()
torch.manual_seed(11)
yg = torch.arange(10, device=device).repeat_interleave(100)
with torch.no_grad():
    raw = ema_G(torch.randn(1000, 100, device=device), yg)
out_of_range = float((raw.abs() > 1).float().mean())
gen = raw.clamp(-1, 1)
idx = [(test_y == c).nonzero().flatten()[:200] for c in range(10)]
f_real = features(test_X[torch.cat([i[:100] for i in idx])])
f_real2 = features(test_X[torch.cat([i[100:] for i in idx])])
f_gen = features(gen)
```

```
print(f'{"comparison":22s}{"FD (CIFAR-CNN)":>16s}{"MMD^2 (CIFAR-CNN)":>19s}'
      f'{"out-of-range":>14s}')
print(f'{"real vs. real":22s}{d2l.fid_score(f_real2, f_real):16.2f}'
      f'{d2l.kid_score(f_real2, f_real):19.3f}{"--":>14s}')
print(f'{"generated vs. real":22s}{d2l.fid_score(f_gen, f_real):16.2f}'
      f'{d2l.kid_score(f_gen, f_real):19.3f}{out_of_range:14.3f}')
# Variance within each class, averaged over features and classes
cvar = lambda f: float(np.asarray(f, np.float64).reshape(10, 100, -1)
                       .var(1).mean())
print(f'within-class feature variance: generated {cvar(f_gen):.3g}, '
      f'real {cvar(f_real):.3g}, second real set {cvar(f_real2):.3g}')
```

```
comparison              FD (CIFAR-CNN)  MMD^2 (CIFAR-CNN)  out-of-range
real vs. real                     0.68             -0.007            --
generated vs. real                2.31              0.119         0.007
within-class feature variance: generated 0.17, real 0.214, second real set 0.207
```

In FD the generated set lies two to four times as far from the real test images as the second real set does. In MMD^2 the real-versus-real estimate fluctuates around zero, while the generated-versus-real estimate is small and positive. The variance line shows what the distances cannot separate: the generated images have about 20% less within-class feature variance than the real ones, while a second real set comes within 5%. The generator has lost some of the variety within each class, and alignment cannot see that loss.

16.6.4.4 Two Baselines

Two constructed generators show what each measurement can and cannot see. The first keeps the same 1,000 generated images and shifts their *requested* labels cyclically, the counterexample from the start of this section. The second is a degenerate generator that returns, for every request of class c, the real test image that the referee classifies as c most confidently. FD and MMD^2 never see a label, so for the first baseline the values of the previous cell stand; we print only the alignment.

```
y_perm = (yg + 1) % 10                          # same images, shifted requests
balanced = torch.cat([i[:100] for i in idx])    # 100 test images per class
with torch.no_grad():
    pred_gen = cnn(gen).argmax(1)
    # The referee's probability of the true class, per class and image
    p_true = cnn(test_X[balanced]).softmax(1)[
        torch.arange(1000, device=device), test_y[balanced]].reshape(10, 100)
    most_confident = balanced.reshape(10, 100)[
        torch.arange(10, device=device), p_true.argmax(1)]
    fixed = test_X[most_confident].repeat_interleave(100, dim=0)
    pred_fix = cnn(fixed).argmax(1)
print(f'permuted labels: alignment '
      f'{float((pred_gen == yg).float().mean()):.2f} -> '
      f'{float((pred_gen == y_perm).float().mean()):.2f}')
print(f'most confident image per class: alignment '
      f'{float((pred_fix == yg).float().mean()):.2f}, '
      f'within-class variance {cvar(features(fixed)):.3g}')
```

```
permuted labels: alignment 0.74 -> 0.02
most confident image per class: alignment 1.00, within-class variance 0
```

Shifting the requests drops alignment from about 0.7 to a few percent, while FD and MMD^2 cannot change, because neither receives a label. The confident-image generator reaches alignment 1.00 with zero within-class variance: it beats real images on alignment while returning one image per request. This *within-class collapse* is mode collapse inside each conditional distribution. The conditional objective Equation 16.41 penalizes it, because a point mass is far from $p(\cdot \mid c)$, and so does the within-class variance; alignment rewards it.

Conditional evaluation thus has three axes, fidelity, diversity and alignment, and each of our measurements sees only part of them. Published scores combine them: the Fréchet joint distance embeds image and condition together (DeVries et al. 2019), and the classification accuracy score trains a classifier on generated pairs and tests it on real ones (Ravuri and Vinyals 2019). Exercise 4 extends our diagnosis to per-class distances, and Exercise 5 uses the same instruments on BigGAN's truncation trick, which trades variety for fidelity on purpose.

We can now ask a GAN for a class and check what we got: whether it obeys, how much it varies, and how close it comes to the data. Pix2pix has a paired target for every input and still needs its critic. Why a learned comparison helps when a regression target exists is the question of Section 16.7.

16.6.5 Summary

Conditional generation gives the request to both players and plays the chapter's game on sample–condition pairs. Its optimal critic is the per-class log density ratio $\log(p(x \mid c)/q(x \mid c))$ and its value averages the per-class divergences, so it detects perfect images with the wrong labels, which the unconditional game cannot. With same-class pairs the pairing loss plays the game of Section 16.3 within every class, up to one free constant per class. Bayes' rule splits the optimal critic into a label term and a realness term, which gives the projection critic $D(x, c) = e_c^\top \phi(x) + \psi_0(x)$. On CIFAR-10 the referee recognizes the requested class in about 70% of the generated images, against 78% of real images, and the generated images have about a fifth less within-class variety than real data. Alignment, feature distances and

within-class variance each catch a different failure.

16.6.6 Exercises

1. code **Two slices with known densities.** Let $c \in \{1, 2\}$ be equally likely, $p(\cdot \mid 1) = \mathcal{N}(-2, 1)$ and $p(\cdot \mid 2) = \mathcal{N}(2, 1)$, and let the generator be wrong on one slice only: $q(\cdot \mid 1) = \mathcal{N}(0, 1)$ and $q(\cdot \mid 2) = \mathcal{N}(2, 1)$.
   1. Compute the value Equation 16.41 by numerical integration on a grid.
   2. Train a small critic on x and a one-hot encoding of c by minimizing $-\frac{1}{2}V(D)$ of Equation 16.39 on samples, and compare the converged loss with $-\frac{1}{2}$ times the value from (a). Plot the critic against Equation 16.40 for each condition. How large is it on the matched slice $c = 2$?
   3. Let the generator swap the labels instead, $q(\cdot \mid 1) = \mathcal{N}(2, 1)$ and $q(\cdot \mid 2) = \mathcal{N}(-2, 1)$. Compute the conditional value and the unconditional value with the labels discarded, and check both by retraining the critic.
2. **Separate feature maps.** The derivation of Equation 16.44 modeled both label posteriors with one shared feature map ϕ. Suppose instead that $p(c \mid x) \propto \exp(u_c^\top \phi_p(x))$ and $q(c \mid x) \propto \exp(w_c^\top \phi_q(x))$ with $\phi_p \neq \phi_q$.
   1. Derive the critic that takes the place of Equation 16.44, and show that it is a projection head over the concatenated feature map $(\phi_p(x), \phi_q(x))$.
   2. Let there be K classes. Show that every function $D(x, c)$ can be written in the form Equation 16.44 once ϕ has at least K dimensions. How many dimensions does ϕ have in `ProjectionCritic`?
   3. Given (b), when does the projection form restrict the set of critics? What does the shared-map assumption then restrict in the CIFAR-10 experiment?
3. [short-code] **The critic as a classifier.** The trained critic `net_D` assigns each image ten scores, one per class.
   1. Use Equation 16.43 to express $D^\star(x, c) - D^\star(x, c')$ through the real and the generated label posteriors. How does $\arg\max_c D^\star(x, c)$ differ from the rule $\arg\max_c p(c \mid x)$ of the referee?
   2. For the test images indexed by `bal`, compute $S_{ic} = D(x_i, c)$ for all ten classes. Report the fraction of images that score higher under their own label c than under $c+1$, and the accuracy of $\arg\max_c S_{ic}$. Then estimate the per-class constant $\hat{\zeta}_c$ as the critic's mean score on the generated images `gen` of class c (labels `yg`), and classify with $\arg\max_c(S_{ic} - \hat{\zeta}_c)$. Expect between a quarter and a third of the images to be classified correctly, two to three times chance. Why does the raw rule do worse?
   3. Why must the corrected rule fall to chance as the generator improves? If the first fraction in (b) fell to one half after longer training, name two states of the game that would produce this, and a measurement of this section that tells them apart.
4. code **Per-class distances.** The features `f_gen`, `f_real` and `f_real2` hold 100 images of each class, in class order.
   1. A covariance matrix estimated from 100 samples of 128-dimensional features has rank at most 99. How does this affect a per-class Fréchet distance, and why does the unbiased MMD^2 estimator not suffer in the same way?
   2. For each class, compute MMD^2 between the generated and the real features, and between the two real sets. Rank the classes by the difference and compare with the ranking by alignment; display the samples of a class on which the two rankings disagree.

3. A conditional generator may ignore the condition, obey it through shortcuts such as background or palette, collapse within a condition, leak between conditions, or neglect a rare condition. Which of these failures does each per-class diagnostic (alignment, feature variance, MMD^2) detect? Check your answer on the collapse baseline.

5. code **Truncated latents.** The truncation trick of Brock et al. (2019) draws each latent coordinate from a standard Gaussian truncated to $[-\tau, \tau]$. Such a coordinate has variance $1 - \sqrt{2/\pi}\,\tau e^{-\tau^2/2} / \operatorname{erf}(\tau/\sqrt{2})$.
   1. For the 100-dimensional latent code of `CondGenerator` and $\tau \in \{0.5, 1, 2\}$, compute the probability that an untruncated draw lies inside $[-\tau, \tau]^{100}$ and the typical norm of a truncated draw. Compare it with the norm of the training latents.
   2. Implement truncated sampling for `ema_G` and sweep $\tau \in \{0.5, 1, 2, \infty\}$, where $\tau = \infty$ is ordinary sampling. Report the alignment, the within-class feature variance and the Fréchet distance to `f_real`, and display the samples.
   3. BigGAN found that only 16% of its models trained without orthogonal regularization tolerated truncation without saturation artifacts, against 60% with it; `ema_G` had none. At which τ do your measurements change most, given the norms from (a)? What does this say about how the generator treats latent codes with norms it never saw in training?

   Adapted from Simon Prince, Understanding Deep Learning, Problem 15.6.

6. code **A rare class.** ▣ Keep only 50 of the 5,000 automobile training images and all images of the other classes, so that $p(c_{\text{rare}}) = \epsilon = 50/45{,}050 \approx 0.0011$.
   1. Show that a generator that serves the rare slice arbitrarily badly raises Equation 16.41 by at most $2\epsilon \log 2$. Class-balanced sampling of the real batches raises the weight of the slice to $1/10$. Which risk does it introduce, given that the slice holds only 50 images?
   2. Train with `d2l.train_r3gan` for 4,000 steps on real batches drawn uniformly from the reduced set, and on class-balanced real batches, whose labels then condition the generator uniformly; as a reference, train at the same budget on the full set. For the rare class, report the alignment, the per-class MMD^2 of problem 4b, and the feature distance of each generated sample to its nearest neighbor among the 50 training images. Which scheme serves the rare class better, and what does it cost the other classes?

7. **Auxiliary classifier bias.** ▣ An auxiliary classifier critic (Odena et al. 2017) contains a classifier $C(c \mid x)$, and the generator also minimizes the classification loss $\mathbb{E}_{c\sim p(c),\, x\sim q(\cdot\mid c)}[-\log C(c \mid x)]$ of its own samples. Let $q(x)$ and $q(c \mid x)$ be the image marginal and the label posterior of the generated pairs.
   1. Show that this loss equals
   $$\mathbb{E}_{x\sim q}\Big[H(q(\cdot \mid x)) + D_{\text{KL}}(q(\cdot \mid x) \,\|\, C(\cdot \mid x))\Big],$$
   where H is the entropy.
   2. Hold C fixed, set $q(x) = p(x)$, and let $q(\cdot \mid x)$ vary freely at every x, ignoring the constraint that the generated labels follow $p(c)$. Which label posterior minimizes the loss? With $C = p(c \mid x)$ and two overlapping classes, show that the generator can lower the loss below its value at $q = p$.
   3. Take the expectation of the label term $\log[p(c \mid x)/q(c \mid x)]$ of Equation 16.43 under the generated pairs. Which term of the loss in (a) does its denominator cancel?

8. code **The critic at a fixed budget.** ▣ The run above fixes the penalty weight $\gamma = 0.5$

and the projection head.

1. Train for 4,000 steps at $\gamma \in \{0.05, 0.5, 5\}$, and report the logged squared input gradients (`history['R1'] + history['R2']`), the alignment, and the within-class feature variance. Which change by an order of magnitude, and which barely move? Since γ is a squared length in pixel space, why does CIFAR-10 need a different value from the sprites?
2. Replace the projection head of `ProjectionCritic` by concatenation: append the one-hot label to the image as ten constant 32×32 planes and score with the head ψ_0 alone. Train both critics with `d2l.train_r3gan` for 4,000 steps with two seeds each. Compare the alignment, the feature distances, and the critic's accuracy as a classifier with its constants removed (problem 3b).
3. By problem 2b, neither critic restricts the function $D(x, c)$ for ten classes. What can then explain a difference at a fixed budget?

16.7 Adversarial Losses Beyond GANs

Almost every regressor in this book was trained with squared error, and for house prices or class probabilities that was the right choice. Ask a network to enlarge a thumbnail, though, and squared error fails. Many sharp photographs shrink to the same thumbnail: the same face with its hair falling slightly differently, the same lawn with its blades of grass elsewhere. Squared error rewards the prediction closest to all of them on average, their pixelwise mean, and that mean is a blur that matches none of them. The critic of this chapter asks a different question of a prediction: whether it looks like a sample from the data. Pix2pix, in the previous section, had a paired target for every input and still added a critic to its L_1 loss. This section asks why squared error averages, and why a learned comparison does not.

The question has outlived stand-alone GANs. Most image, video and audio generators are now trained by regression, without a two-player game, yet an adversarial loss still sits inside the one-step generators distilled from them, inside the autoencoders many of them run on, and inside many speech vocoders. We work out the averaging on a problem with one input and two answers, follow the argument into these systems, and end with what the repairs of this chapter leave unsolved.

16.7.1 Conditional Means and Ambiguity

Many generation problems are conditional predictions. A super-resolution network receives a low-resolution image c, the condition, and must produce a high-resolution image x, the sample. A vocoder receives a spectrogram and must produce a waveform. A one-step student receives a noise vector and must produce, in one forward pass, the image its teacher builds over many network evaluations. Before comparing losses for such problems, we ask what the usual loss rewards.

16.7.1.1 The Conditional-Mean Decomposition

Write g for the predictor and $\mu(c) = \mathbb{E}[x \mid c]$ for the conditional mean. Adding and subtracting $\mu(c)$ inside the square produces a cross term $2\,\mathbb{E}\big[(x - \mu(c))(\mu(c) - g(c))\big]$, which vanishes: given c, the second factor is fixed and the first has mean zero. What remains is

$$\mathbb{E}\big[(x - g(c))^2\big] \;=\; \mathbb{E}\big[(x - \mu(c))^2\big] \;+\; \mathbb{E}\big[(\mu(c) - g(c))^2\big]. \tag{16.45}$$

The first term is the spread of x around its mean, which no predictor can remove. The second is the squared distance from $g(c)$ to $\mu(c)$, so squared error grades a predictor by its distance to the conditional mean, and the best predictor is $g = \mu$. When $p(x \mid c)$ concentrates on one answer, the mean is that answer. When it spreads over several, the mean averages them and need not be an answer itself: the mean of many sharp images that share a thumbnail is not sharp.

An adversarial loss grades a prediction by a learned comparison instead. A critic $D(x, c)$ is trained by logistic regression to tell data pairs $(x, c) \sim p$ from pairs $(\hat{x}, c) \sim q$ formed with the predictor's output. As Section 16.1 showed, the best such critic is the log density ratio

$$D^\star(x, c) = \log \frac{p(x, c)}{q(x, c)}.$$

It is strongly negative where predictions land with no data near them. The predictor raises the critic's score of its own outputs by minimizing $-\log \sigma(D(\hat{x}, c))$, the non-saturating loss of Section 16.1, so a prediction in an empty region between valid answers scores low, however close it is on average to all of them. The two losses agree when $p(x \mid c)$ has one answer the predictor can reach. When it has several, squared error favors their mean and the critic favors a prediction among the data.

16.7.1.2 Two Bands

The smallest problem that separates the two losses has one input, one output and two answers. Draw the condition c uniformly from $[-1, 1]$ and set

$$x = c + b + 0.05\,\xi, \qquad b \in \{-1, +1\} \text{ with probability } \tfrac{1}{2} \text{ each}, \quad \xi \sim \mathcal{N}(0, 1), \tag{16.46}$$

with the offset b drawn independently of c. The data form two parallel bands, $x = c \pm 1$, two units apart, forty times the noise width. The conditional mean $\mu(c) = c$ runs through the empty corridor between them, and its expected squared error is $1 + 0.05^2 \approx 1.0025$, the first term of Equation 16.45. A predictor that follows the band $x = c + 1$ is right for half of the data and two units off for the other half, so its expected squared error is $\frac{1}{2} \cdot 4 + 0.05^2 \approx 2.0025$. Squared error prefers the corridor, where no data point lies.

Both students are the same deterministic network, a perceptron with two hidden layers of 32 tanh units that maps c to one prediction $\hat{x} = g(c)$, and both start from the same weights. The critic is a perceptron of the same size with leaky ReLU units that scores pairs (x, c). All three come from `d2l.make_mlp` of Section 16.1. Since the data distribution is known, `make_bands` draws a fresh batch at every training step.

```
def make_bands(n):
    """Draw n pairs (c, x) from the two bands x = c + b + 0.05 xi."""
    c = torch.rand(n, 1) * 2 - 1
    b = torch.randint(0, 2, (n, 1)) * 2.0 - 1.0
    return c, c + b + 0.05 * torch.randn(n, 1)

torch.manual_seed(0)
g_mse = d2l.make_mlp([1, 32, 32, 1], nn.Tanh())
```

```
g_adv = d2l.make_mlp([1, 32, 32, 1], nn.Tanh())
g_adv.load_state_dict(g_mse.state_dict())  # both students start alike
D = d2l.make_mlp([2, 32, 32, 1], nn.LeakyReLU(0.2))
c_data, x_data = make_bands(200)  # a data sample for the plot
```

The first student minimizes squared error with Adam. Its learning rate decays along a cosine from 0.01 to zero over 2,000 steps, so the last steps average out the minibatch noise and the curve settles where Equation 16.45 says it should.

```
def train_mse(g, num_steps=2000, batch_size=128, lr=0.01):
    opt = torch.optim.Adam(g.parameters(), lr=lr)
    sched = torch.optim.lr_scheduler.CosineAnnealingLR(opt, num_steps)
    for _ in range(num_steps):
        c, x = make_bands(batch_size)  # a fresh batch at every step
        opt.zero_grad()
        ((g(c) - x) ** 2).mean().backward()
        opt.step()
        sched.step()

train_mse(g_mse)
```

The second student plays the log-loss game, with the losses `d2l.gan_loss_D` and `d2l.gan_loss_G` of Section 16.4. At each step the critic takes one logistic-regression step on real pairs (x, c) against predicted pairs $(\hat{x}, c)$, and then the student takes one step on $-\log \sigma(D(\hat{x}, c))$ against the updated critic. The prediction enters the game with the data's observation noise, $\hat{x} = g(c) + 0.05\,\xi'$ with a fresh $\xi' \sim \mathcal{N}(0, 1)$, so the two distributions differ in where they put their mass, not in how smooth they are.

```
def train_adversarial(g, D, num_steps=8000, batch_size=128):
    opt_G = torch.optim.Adam(g.parameters(), lr=1e-3)
    opt_D = torch.optim.Adam(D.parameters(), lr=2e-3)
    for _ in range(num_steps):
        c, x = make_bands(batch_size)
        x_hat = g(c) + 0.05 * torch.randn(batch_size, 1)  # the data's noise
        real, fake = torch.cat([c, x], 1), torch.cat([c, x_hat], 1)
        opt_D.zero_grad()
        d2l.gan_loss_D(D, real, fake.detach()).backward()
        opt_D.step()
        opt_G.zero_grad()
        d2l.gan_loss_G(D, real, fake).backward()
        opt_G.step()

train_adversarial(g_adv, D)
```

Three numbers summarize each student on a grid of 201 inputs: the mean distance from its curve to the nearer band, the mean distance to the conditional mean $x = c$, and the fraction of inputs at which the curve is more than 0.25 from both bands.

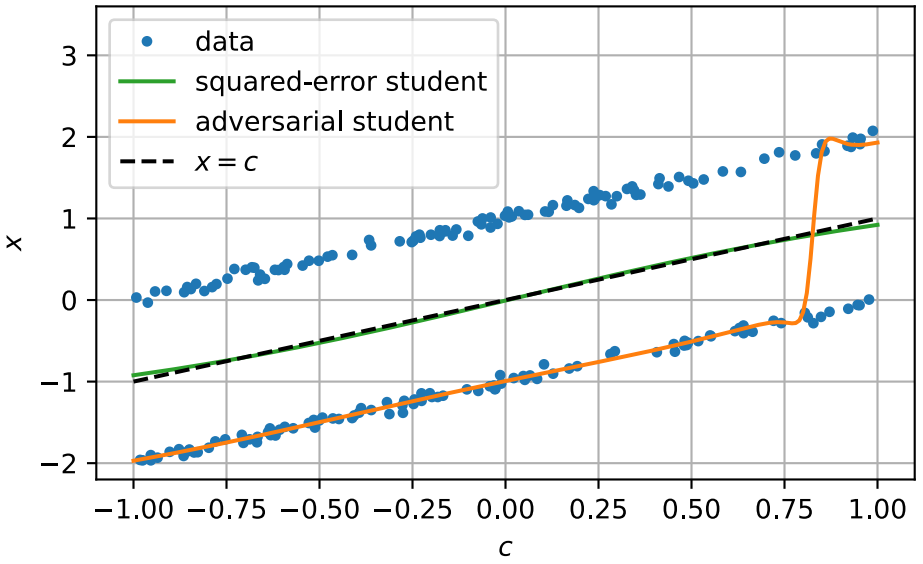
```
cs = torch.linspace(-1, 1, 201).reshape(-1, 1)
with torch.no_grad():
```

```
    curves = {'squared error': g_mse(cs), 'adversarial': g_adv(cs)}
for name, x_hat in curves.items():
    to_band = torch.minimum((x_hat - cs - 1).abs(), (x_hat - cs + 1).abs())
    print(f'{name}: {to_band.mean():.2f} from the nearer band, '
          f'{(x_hat - cs).abs().mean():.2f} from x = c, '
          f'{(to_band > 0.25).float().mean():.0%} off the bands')
```

```
squared error: 0.98 from the nearer band, 0.02 from x = c, 100% off the bands
adversarial: 0.02 from the nearer band, 0.99 from x = c, 2% off the bands
```

The plot draws both curves over a sample of the data, with the conditional mean dashed.

```
d2l.plot([c_data, cs, cs, cs],
         [x_data, curves['squared error'], curves['adversarial'], cs],
         '$c$', '$x$', ['data', 'squared-error student',
                        'adversarial student', '$x = c$'],
         ylim=(-2.2, 3.6), fmts=('C0.', 'C2-', 'C1-', 'k--'),
         figsize=(5.5, 3.2))
```



The squared-error student lies on average within about 0.02 of the dashed line $x = c$, a little farther at the ends of the interval, as Equation 16.45 predicts, and so about one unit from every band: everywhere it predicts a value that never occurs in the data. The adversarial student lies on a band, a few hundredths from it on average, and leaves the bands only at zero, one or two narrow crossings; other seeds change which band it follows and where it crosses, not these numbers. A curve that changes bands must pass through the corridor, since it is continuous. The adversarial loss confines that passage to a narrow interval, whereas squared error places every prediction in the corridor.

16.7.1.3 Ambiguity and Capacity

The adversarial student does not recover the conditional distribution: a deterministic map gives one answer per input, so following one band leaves the other unrepresented. A larger network would not help. The condition c carries no information about the band, so no deterministic function of c, however large, picks the right band more than half the time. The missing ingredient is information. Given a latent bit $z \in \{-1, +1\}$ drawn alongside c, a student can send each value of z to one band and so produce both (Exercise 7). Generators that must produce varied outputs therefore take a latent input or keep a multi-step sampler, and they need a loss that does not reward piling every sample onto one band, the question of mode coverage that Section 16.3 studies.

Averaging has two sources, and only one of them yields to a larger model. In the first, the input does not determine the output: the thumbnail, the two bands, and, below, the phase of a waveform. In the second, the input determines the output but the model cannot compute the map in one pass. The noise vector of a one-step student fixes its teacher's image, but a student that cannot reproduce the teacher's long computation sees several inputs as alike. Either way the best squared-error fit averages over the outputs the model cannot tell apart, and more capacity removes only the second source.

A predictor that samples $p(x \mid c)$ exactly has twice the expected error of the conditional mean, 3 dB less peak signal-to-noise ratio (Exercise 1). Blau and Michaeli (2018) showed that the conflict is general: low distortion and outputs distributed like the data cannot both be reached, so a realistic output must be judged by whether it looks like data, not by its squared error.

16.7.2 Adversarial Terms in Current Systems

The argument predicts where an adversarial term should survive once generators are trained by regression: wherever the output is ambiguous given what the model sees, and the model must answer in one pass. We follow that prediction into three kinds of systems.

16.7.2.1 Denoisers as Conditional Means

Diffusion and flow models generate a sample by repeatedly evaluating a learned denoiser, often dozens or hundreds of times (Ho et al. 2020; Lipman et al. 2023). The denoiser receives a noisy image x_t, a clean image x_0 with noise added at level t, and is trained by squared error to predict x_0. By Equation 16.45 with $(c, x) = (x_t, x_0)$, it learns the conditional mean $\mathbb{E}[x_0 \mid x_t]$. At low noise, x_t nearly determines x_0 and the mean is sharp. At high noise, many clean images are compatible with x_t, and the mean is their blurred average, the corridor of the two bands. A many-step sampler takes only a small step toward that mean, adds fresh noise, and asks again at a lower noise level, so no single average becomes the output. A one-step student has no later steps. If it is trained by squared error toward the teacher's denoised estimate at high noise, it inherits the average.

16.7.2.2 One-Step Distillation

Distillation trains a student to approximate the teacher's distribution in one to four network evaluations. Adversarial diffusion distillation (ADD) trains a one-step student with two losses (Sauer, Lorenz, et al. 2023): a regression toward a frozen teacher, which denoises a noised copy of the student's output, and an adversarial loss against real images, with a critic

built from frozen DINOv2 features and small trainable heads. In the reported one-step ablation, with all other settings fixed, the regression alone gives FID 315.6, the adversarial loss alone 20.8, and both together 20.6. A student fitted only to a conditional-mean target does not produce usable samples. The adversarial term supplies the sample quality, and the distillation term leaves FID nearly unchanged while raising the CLIP score, the agreement of the image with its prompt.

Distribution matching distillation (DMD) replaces the critic by two models trained by regression (Yin, Gharbi, Zhang, et al. 2024). It moves each generated sample x' along the difference of two scores, the gradients of log densities, $\nabla_x \log p(x') - \nabla_x \log q(x')$, where q is now the student's own distribution. Two diffusion models supply the scores on noised copies of x': the frozen teacher for the data, and a second model, trained online on the student's samples, for q. At noise level t the difference is $\nabla_x \log(p_t/q_t)$, the input gradient of the optimal log-loss critic between the noised data p_t and the noised samples q_t. DMD therefore follows the direction a perfect critic would give, without a classifier. The loop remains coupled, since the second model trains alongside the student, but neither network maximizes what the other minimizes. Its successor DMD2 adds a critic against real images (Yin, Gharbi, Park, et al. 2024).

16.7.2.3 Tokenizers and Vocoders

Many image generators run on an autoencoder that compresses each image into a shorter code. Widely used ones, from VQGAN to the Stable Diffusion autoencoder, add a critic that scores image patches, because the code does not fix the exact texture at each location and pixelwise losses average the plausible textures into a blur (Esser et al. 2021; Rombach et al. 2022). ViTok-v2 replaced the adversarial loss with a fixed perceptual loss on self-supervised DINOv3 features when it scaled its autoencoder to five billion parameters, reporting that adversarial losses prevented stable scaling (Hansen-Estruch et al. 2026). A perceptual loss is still squared error, but on features that change little across the textures the code leaves open, so averaging them does not blur the pixels.

Speech vocoders face the same ambiguity in phase. A spectrogram fixes the frequency content and the magnitude of each frame but not its phase. If a frame is $\sin(\omega t + \vartheta)$ with the phase ϑ uniform on $[-a, a]$, the squared-error prediction, the average over ϑ, is a sine of the same frequency whose amplitude has shrunk by the factor $\sin a/a$ (Exercise 4). With the phase entirely unknown, $a = \pi$, the prediction is silence. A critic that sees the waveform rejects the shrunken frame, which is why HiFi-GAN and the BigVGAN family train against critics (Kong et al. 2020; Lee et al. 2023), and why RAF, a universal speech synthesizer, trains with the paired loss of Section 16.3 (Lee and Choi 2026). Whether fixed perceptual features can replace the critic for audio, as they did for ViTok-v2, is open.

16.7.3 The Objective without the Game

Every system above that keeps a critic inherits the training problems of the earlier sections, and ViTok-v2 shows that they can block scaling. Methods respond in three ways, and the third separates the adversarial objective from the game that usually optimizes it.

16.7.3.1 Responses to Instability

The first response regularizes the dynamics of the game: zero-centered penalties on the critic's input gradient make training converge near the equilibrium (Mescheder et al. 2018),

as Section 16.4 derives. The second constrains the critic, by bounding its slope, by freezing its features as ADD does, or by augmenting both of its inputs. The third removes the game: a fixed discrepancy such as MMD, or, in diffusion and flow models, a regression on a closed-form target with a unique minimizer, so that training needs no equilibrium between two networks. Figure 16.9 compares the three.

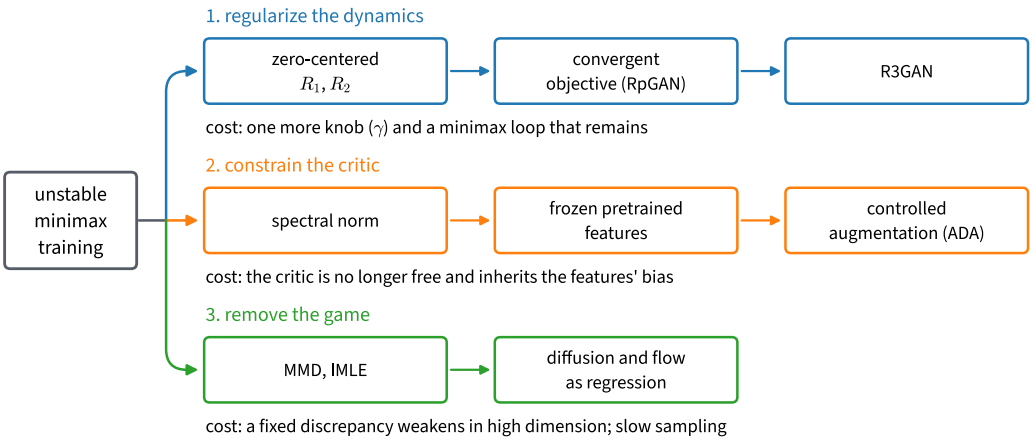


Figure 16.9: Three responses to unstable adversarial training, corresponding to different sources of instability: regularize the dynamics of the game, constrain the critic, or replace the game with a fixed discrepancy or regression objective. The first retains a minimax loop and a penalty weight; the second inherits the biases of its fixed features; the third may use discrepancies that weaken in high dimensions or require multi-step sampling. Adversarial losses can reappear when non-adversarial models are distilled to a few steps.

16.7.3.2 Likelihood Models as Critics

Standard adversarial training estimates $\log(p/q)$ with a separate network because the generator's density q cannot be evaluated. A likelihood model can evaluate its density, and so it can compute the ratio between two versions of itself. Direct discriminative optimization (DDO) (K. Zheng, Chen, Chen, et al. 2025) takes a pretrained likelihood model q_θ, such as a diffusion model, compares real data with samples from a frozen reference copy q_{ref}, and parameterizes the critic as

$$D_\theta(x) = \log\left(q_\theta(x)/q_{\text{ref}}(x)\right). \tag{16.47}$$

For a diffusion model, which offers only a bound on its log-likelihood, the ratio is estimated from the difference of the two models' denoising losses. Training this critic by logistic regression on real against reference samples updates q_θ directly, with no discriminator network and no second player. If training reaches $q_\theta = p$, then $D_\theta = \log(p/q_{\text{ref}})$, the optimal critic between the data and the reference model (Exercise 6), so the parameterization can represent the exact solution of the game it plays.

The adversarial objective thus needs an estimate of $\log(p/q)$ or of its gradient. The convergence failures of Section 16.4 come from estimating it with a second network trained against

the generator. Vocoders and ADD keep that game, ViTok-v2 drops the objective, DMD estimates the gradient with two score models, and DDO represents the ratio with the likelihood model itself.

16.7.3.3 Remaining Limits

The chapter began with a game whose generator could lose its gradient, whose training could circle without converging, and whose samples could collapse onto a few modes. The repairs work. The non-saturating loss keeps a gradient on the samples the critic rejects, and zero-centered penalties keep one when nearby supports separate and make gradient training converge near the equilibrium (Section 16.4). With these, a plain convolutional generator learns to draw images (Section 16.5). Four limits remain. The generator is a sampler without a density, so it cannot report how likely an image is. Training still needs two networks and a game that converges reliably only under a penalty whose weight must be chosen for the data. No likelihood term penalizes a missing mode, and the guarantee against mode-dropping minima holds only for a fixed one-to-one pairing of real and generated samples (Section 16.3). And the game can still become unstable at scale, as ViTok-v2 reported. Diffusion and flow models, the subject of Chapter 17, avoid the game: they are trained by regression against closed-form targets, computed from each training example.

16.7.4 Summary

Squared error measures a prediction's distance to the conditional mean, so where the input leaves the output ambiguous, or the model cannot compute the map in one pass, a regression-trained model predicts an average that need not be a valid output. A critic, whose optimum is the log density ratio, rewards predictions among the data: on two bands the squared-error student runs through the empty corridor, while the adversarial student stays on a band, though as a deterministic map it covers only one. A squared-error denoiser is a conditional mean, so a one-step student fitted to it inherits the average; autoencoders and vocoders face the same ambiguity in texture and phase, and all three keep a critic or a stand-in for one. The adversarial objective needs an estimate of $\log(p/q)$, not a two-player game, as DMD and DDO show.

16.7.5 Exercises

1. **Distortion of a perfect sampler.** Let $\hat{x}$ be drawn from $p(x \mid c)$, independently of x given c, so that the predictor samples the conditional distribution exactly (Blau and Michaeli 2018).
    1. Use Equation 16.45 to show that $\mathbb{E}\big[(x-\hat{x})^2\big] = 2\,\mathbb{E}\big[(x-\mu(c))^2\big]$. How many decibels of peak signal-to-noise ratio, $10\log_{10}\left(x_{\text{max}}^2/\mathbb{E}[(x-\hat{x})^2]\right)$, does the sampler lose against the conditional mean?
    2. On the two bands Equation 16.46, compute the expected squared error of the exact sampler and rank it against the conditional mean (about 1.0025) and the band $x = c + 1$ (about 2.0025). Which of the three can an optimal critic of (x, c) pairs not distinguish from the data?
    3. A deterministic predictor can match the distribution of x without sampling. Let $x \sim \mathcal{N}(0,1)$ and $c = x + \tau\xi$ with $\xi \sim \mathcal{N}(0,1)$ independent of x. Show that $\hat{x} = c/\sqrt{1+\tau^2}$ has the distribution of x, and compute the ratio of its squared

error to that of the conditional mean for $\tau \in \{0.1, 1, 10\}$. When is the extra error small?

2. code **Mean and median.** Absolute error $\mathbb{E}\big[|x - g(c)|\big]$ is minimized by a conditional median, a value t with $P(x \le t \mid c) = \frac{1}{2}$.
   1. Let the offset b of Equation 16.46 take the values -1, 0 and $+1$ with probabilities $(\frac{1}{3}, \frac{1}{3}, \frac{1}{3})$ or $(\frac{1}{4}, \frac{1}{4}, \frac{1}{2})$. Find the conditional mean and median in each case, and say whether each is a valid output.
   2. Return to two bands with $P(b = +1) = r$, and write the conditional mean as a function of r. For $r = \frac{3}{4}$, show that the conditional median lies on the upper band and compute its offset from c to two decimal places. For $r = \frac{1}{2}$, describe the absolute-error risk as a function of $g(c) - c$ inside the corridor.
   3. Train the squared-error student of Section 16.7.1.2 and a copy trained with absolute error on the data of (a) and (b), with three seeds for $r = \frac{1}{2}$, and compare with your predictions. When is the conditional median a valid output?

3. code **Pointwise plus adversarial.** ADD trains its student on the sum of an adversarial loss and a weighted pointwise distillation loss (Sauer, Lorenz, et al. 2023).
   1. Add κ times the squared error against the observed x to the loss of the adversarial student of Section 16.7.1.2. Train with $\kappa \in \{0.1, 1, 10, 100\}$, two seeds each, and report the mean distances to the nearer band and to the conditional mean. Near which κ does the curve leave the bands?
   2. ADD's teacher instead denoises a noised copy of the student's output and uses its estimate as a constant target. Let it see $\tilde{x} = \hat{x} + \tau\xi$, with $\xi \sim \mathcal{N}(0, 1)$ fresh at every step. Show that the posterior mean of the band center given $\tilde{x}$ is $c + \tanh\big((\tilde{x} - c)/(\tau^2 + 0.05^2)\big)$, and find its limits as $\tau \to 0$ and $\tau \to \infty$. Repeat the sweep of
      (a) with this target, gradient stopped, at $\tau = 0.1$ and $\tau = 2$, and relate the result to ADD's one-step ablation (distillation alone FID 315.6, adversarial alone 20.8, both 20.6).

4. **Phase and the conditional mean.** Model one frame of a waveform as $x_t = \sin(\omega t + \vartheta)$ for $t = 0, \dots, N - 1$, where the spectrogram fixes ω and the phase ϑ is uniform on $[-a, a]$ with $0 < a \le \pi$.
   1. Show that the squared-error prediction $\mathbb{E}[x_t]$ has $\sin a / a$ times the amplitude of the valid waveforms, and evaluate the factor at $a = \pi/2$.
   2. Let $\omega = 2\pi k/N$ for an integer k with $0 < k < N/2$. Show that the magnitudes of the discrete Fourier transform of x do not depend on ϑ. Which outputs minimize a loss on these magnitudes, such as the mel-spectrogram term of HiFi-GAN (Kong et al. 2020)?
   3. Concatenate two such frames with independent phases, so that a magnitude loss on each frame is zero. What is wrong with the waveform at the boundary, and what must the input of a critic contain to penalize it?

5. **Frozen critic features.** Projected GAN and ADD score a sample through a frozen pretrained feature map ϕ and train only a head r, so that $D(x) = r(\phi(x))$ (Sauer et al. 2021; Sauer, Lorenz, et al. 2023). Let p_ϕ and q_ϕ be the distributions of $\phi(x)$ under p and q, and let r range over all functions.
   1. Write the log-loss value $V(D) = \mathbb{E}_{x\sim p}[\log \sigma(D(x))] + \mathbb{E}_{x\sim q}[\log \sigma(-D(x))]$ as an

expectation over p_ϕ and q_ϕ and maximize it pointwise over r. What is the optimal head, which divergence does the maximal value measure, and which generators q minimize it?
2. Let $p = \mathcal{N}(0, I_2)$ on $\mathbb{R}^2$ and $\phi(x) = x_1$. Give a minimizer $q \neq p$.
3. Suppose the generator is scored by the Fréchet distance of Section 16.5, computed with the same features ϕ. Which score does a minimizer from (a) reach, and what does it say about q? Relate the answer to the finding of Kynkäänniemi et al. (2023) that ImageNet-pretrained critics can lower Inception-based FID.

6. **DDO's implicit critic.** DDO (K. Zheng, Chen, Chen, et al. 2025) plays the log-loss game between real data and samples of a frozen model q_{ref} with the critic D_θ of Equation 16.47. It minimizes $J(\theta) = -\mathbb{E}_{x\sim p}[\log \sigma(D_\theta(x))] - \mathbb{E}_{x\sim q_{\text{ref}}}[\log \sigma(-D_\theta(x))]$ over the parameters of q_θ, starting from $\theta = \theta_{\text{ref}}$.
   1. Show in three lines that if training reaches $q_\theta = p$, this critic is the optimal critic between p and q_{ref}.
   2. Show that at $\theta = \theta_{\text{ref}}$ the gradient of J is one half of the gradient of the negative log-likelihood $-\mathbb{E}_{x\sim p}[\log q_\theta(x)]$. Use $\mathbb{E}_{x\sim q_\theta}[\nabla_\theta \log q_\theta(x)] = 0$, which follows from differentiating $\int q_\theta(x)\, dx = 1$.

7. code **A latent bit.** ⊠ Give the student of Section 16.7.1.2 a latent input $z \in \{-1, +1\}$, uniform and independent of c, so that it maps (c, z) to $\widehat{x}$. The critic still scores (x, c) pairs.
   1. Train one such student with squared error and one adversarially, and find the band each value of z lands on. Which loss gives the student an incentive to use z?
   2. IMLE (Li and Malik 2018) matches each data point with its nearest generated sample. With a binary latent, its loss is $\mathbb{E}\big[\min_z (g(c, z) - x)^2\big]$. Train a student with it and compare with (a). Which property of the loss forces both latent values into use without a critic?
   3. Now let $P(b = +1) = \frac{3}{4}$. Which band frequencies can a student with a uniform binary latent represent at a given c? Predict, from the Jensen–Shannon divergence to the data, the choice of the adversarial student, and check by training. What must change in the latent input to represent $\frac{3}{4}$?

8. code **Multi-round refinement.** ⊠⊠ In practice DDO minimizes

$$J_{\alpha,\beta}(\theta) = -\mathbb{E}_{x\sim p}\big[\log \sigma(\beta D_\theta(x))\big] - \alpha\, \mathbb{E}_{x\sim q_{\text{ref}}}\big[\log \sigma(-\beta D_\theta(x))\big]$$

with D_θ of Equation 16.47 and β between 0.01 and 0.1, over several rounds, each of which replaces q_{ref} by the previous round's model (K. Zheng, Chen, Chen, et al. 2025).
   1. Let q_θ range over all densities. Minimize the integrand at each x over $D_\theta(x)$ to show that the minimizer is proportional to $q_{\text{ref}}^{1-1/\beta}\, p^{1/\beta}$ and is normalized for exactly one value of α. With $r_n = \log(q_n/p)$ for the model of round n, derive the recursion from r_n to r_{n+1} up to an additive constant. For which β do the rounds converge to p, and what happens at $\beta = 0.05$?
   2. Now let the family q_θ be restricted. Generalize Exercise 6b to show that at $\theta = \theta_{\text{ref}}$ the gradient of $J_{\alpha,\beta}$ is $\beta/2$ times the gradient of the negative log-likelihood, for every α. Conclude that every fixed point of the rounds is a stationary point of the likelihood.

3. Fit the Gaussian family to $p = \frac{1}{2}\mathcal{N}(-2, 0.5^2) + \frac{1}{2}\mathcal{N}(2, 0.5^2)$ by quadrature, from $q_{\text{ref}} = \mathcal{N}(1, 1)$ with $\alpha = 1$, and track the mean and standard deviation over eight rounds for $\beta \in \{1, 0.75, 0.5, 0.05\}$. Compare the threshold with (a). The Gaussian likelihood has one stationary point, so by (b) the rounds can at best reach the maximum-likelihood fit. Which features of the setting of K. Zheng, Chen, Chen, et al. (2025) must differ for their reported gains to be possible?

16.8 Further Reading

The chapter taught a few ideas on problems small enough to check: what a critic estimates, what a ranking critic changes, why the game needs a penalty to converge, how to score images and condition them, and where a learned comparison is still needed. The field produced far more objectives, architectures, scores and systems than these ideas require. This page points to the ones a reader of the chapter is best placed to read next. Each entry states the problem it addresses, what it changes relative to the chapter, and why to read it; the topics follow the order of the chapter, and nothing in the book depends on them.

16.8.1 Objectives and Critic Constraints

Least squares as a divergence. Section 16.2 derived the divergence that the least-squares *critic* loss evaluates. Mao et al. (2017) started from the other end and asked which target codes make the *generator's* loss a known divergence: with the real and fake codes two apart and the generator's target one above the fake code, minimizing the generator's loss minimizes the Pearson χ^2 divergence between $p + q$ and $2q$. In the paper's convention this is $\frac{1}{2}\int (p-q)^2/(p+q)$, symmetric and bounded by one, not the unbounded $\chi^2(p \,\|\, q)$ of Section 37.3.1.2. The zero–one coding of the chapter's code, with the generator's target equal to the real code, falls outside the derivation; its critic still evaluates the square row of the chapter's table. The derivation takes a few lines in the template of that section, and it shows how much the coding constants matter.

Output activations for f-GAN critics. The variational bound of Section 16.2 asks the critic's output t to lie where the conjugate $f^*(t)$ is finite. Nowozin et al. (2016) meet this with an output activation chosen per divergence. The forward-KL conjugate is finite on all of $\mathbb{R}$ and needs none; the reverse-KL conjugate is finite only for $t < 0$, enforced by $-e^{-v}$ on the network's raw output v; the Jensen–Shannon conjugate is finite for $t < \log 2$, enforced by $\log 2 - \text{softplus}(-v)$; and the original GAN's conjugate is finite for $t < 0$, enforced by $-\text{softplus}(-v)$, the log-sigmoid that Section 16.1 already uses. Their table is the place to look before training a divergence that the chapter did not. A trained critic certifies only a lower bound, and a minibatch estimate of that bound can flatter it: an overshooting critic is penalized only where q is small, which minibatches rarely sample (Exercise 7 of Section 16.2 measures both on the chapter's three-mode mixture).

Three ways to a Lipschitz critic. The Wasserstein-1 game needs a 1-Lipschitz critic, and no network is 1-Lipschitz by construction. Weight clipping (Arjovsky et al. 2017) and spectral normalization (Miyato et al. 2018) bound the product of the layers' Lipschitz constants, a bound that holds everywhere but is often loose. The one-centered gradient penalty (Gulrajani et al. 2017) constrains the gradient only at sampled points, tightly there and not at all elsewhere. These guarantees decide how close a trained critic can come to the Kantorovich potential; Section 16.4 explains why zero-centered penalties replaced the one-centered one.

The relativistic average critic. Jolicoeur-Martineau (2019) proposed, besides the pairing

loss of Section 16.3, a cheaper variant that compares each sample's score with the mean score of the other batch: a real sample against the average generated score, and a generated sample against the average real score. The comparison makes the loss shift-invariant, like the pairing loss, but each sample is still judged against one number, a threshold rather than a rank. The super-resolution network ESRGAN trains with it (Wang et al. 2018), so it is the variant a reader most often meets in image-restoration code.

The relativistic divergence for general payoffs. Section 16.3 computed the value of the pairing game for the logistic payoff, for which the best pair critic is a difference of single-sample scores. For other payoffs it is not, and the value has no closed form. Jolicoeur-Martineau (2020) shows that the pairing objective is still a divergence: its calibrated value is nonnegative and vanishes exactly at $q = p$ for every concave payoff ℓ with $\ell(0) = 0$, $\ell'(0) \neq 0$, and a positive supremum attained or approached at a positive argument. Her normalization is twice the chapter's, so her divergence is $2\, d_{\mathrm{Rp}}$. Read it for the general conditions and for the relativistic least-squares and hinge losses, whose payoffs, shifted to vanish at zero, satisfy them.

16.8.2 Architectures and Training Devices

Architectural devices, 2015 to 2021. Between the 2015 DCGAN network and the plain penalized backbone of Section 16.5, most progress on image GANs came from devices added to the architecture, each aimed at one failure. Progressive growing starts both networks at 4×4 and adds layers during training, against instability at high resolution (Karras et al. 2018). Spectral normalization divides each weight matrix by its largest singular value, which bounds the critic's steepness everywhere, where the penalties of Section 16.4 act only at the samples (Miyato et al. 2018). Self-attention gives both networks image-wide receptive fields for long-range structure (Zhang et al. 2019). BigGAN scales batches and widths and finds that collapse is delayed, not removed (Brock et al. 2019). StyleGAN adds a mapping network and per-layer style modulation to disentangle latent factors (Karras et al. 2019); StyleGAN2 replaces the normalization that caused droplet artifacts by weight demodulation (Karras, Laine, et al. 2020); StyleGAN3 removes texture that sticks to pixel coordinates by band-limited resampling (Karras et al. 2021). Some impose by architecture what the penalties impose through the loss; reading them against R3GAN shows which ones a well-behaved loss makes unnecessary.

Pretrained features in the critic. Projected GAN discriminates in the feature pyramid of a frozen pretrained classifier and reports reaching previous best FID values up to forty times faster (Sauer et al. 2021). The frozen features change what the critic can estimate: it sees the log ratio only through features trained for another task. StyleGAN-T keeps the construction but uses self-supervised features, partly to reduce the overlap with the network that computes FID (Sauer, Karras, et al. 2023); the second entry under Evaluation explains why that overlap matters.

Training at scale. Large runs keep R3GAN's recipe but schedule its constants over a burn-in of 20 million images instead of fixing them (Huang et al. 2024). The learning rate decays from $2 \cdot 10^{-4}$ to $5 \cdot 10^{-5}$ and the penalty weight γ falls tenfold with it, from 0.05 to 0.005 on CIFAR-10 and from 150 to 15 on FFHQ at 256×256. Adam's β_2 rises from 0.9 to 0.99, and the EMA half-life grows from zero to millions of images. The same paper reports that mixed-precision training fails with FP16 and works with BFloat16. It also rebuilds StyleGAN2 on FFHQ at 256×256 at about the same parameter count and budget. Removing StyleGAN2's ten devices, from the mapping network and weight demodulation to lazy regularization,

raises the reported FID from 7.5 to 12.5. The pairing loss with R_1 and R_2 brings it to 11.7, a modern residual backbone to 10.0, and grouped convolutions with inverted bottlenecks to 7.0, below the starting point.

The cost of the penalty. An input-gradient penalty needs a second backward pass through the critic. StyleGAN2 computes R_1 only once every 16 minibatches, with its weight scaled up to compensate, and calls this lazy regularization (Karras, Laine, et al. 2020). StyleGAN2 also trains well on faces with R_1 alone, so the StackedMNIST ablation of Section 16.4, where R_1 alone diverged, does not settle whether one penalty suffices. Adversarial post-training of a video diffusion transformer could not take the second backward pass at all, because its fused attention kernels and sharded training did not support it. It penalizes instead the squared change of the critic's output when a real sample is perturbed by small Gaussian noise, which for small noise approximates the R_1 integrand scaled by the noise variance; without this term its training collapsed (Lin et al. 2025). A reader who needs the penalty at a large scale will find the trade-off between its cost and its frequency in these papers.

Training on limited data. Section 16.5 reads overfitting from the critic's train–holdout gap and argues that an augmentation must be applied to both inputs of the critic and must not map two distributions to one. Differentiable augmentation applies the same random color, translation and cutout transformations to real and generated images in both updates, inside the generator's gradient path (Zhao et al. 2020). Adaptive discriminator augmentation adds two things (Karras, Aittala, et al. 2020). It characterizes which augmentations are non-leaking, applied with probability below one so that the clean distribution can be recovered, and it sets that probability during training from an overfitting statistic, the mean sign of the critic's scores on training images, $r_t = \mathbb{E}[\text{sign } D(x_{\text{train}})]$. The statistic presumes a meaningful zero level of the critic's score, which a pairing critic does not have, since adding a constant to its scores leaves its loss unchanged; with the loss of Section 16.3 a pairwise statistic is needed instead. With adaptive augmentation StyleGAN2 trains on a few thousand images, and the two papers are the place to start for a dataset that small.

16.8.3 Evaluation

Fidelity and coverage separately. FID combines two failures in one number: generated samples that look wrong, and data modes that no generated sample covers. The precision and recall of Kynkäänniemi et al. (2019) separate them through neighborhoods in feature space. Precision is the fraction of generated samples that fall inside the estimated support of the real features, recall the fraction of real samples inside the support of the generated ones. A generator that collapses to one good image keeps a high precision while its recall falls to zero, which FID reports only as a worse score. R3GAN reports recall next to FID for this reason.

The feature network behind the score. FID and KID are only as good as the features they compare. Kynkäänniemi et al. (2023) show that Inception features respond strongly to ImageNet class information, so a generator can lower FID by matching the distribution of ImageNet classes in its samples without improving them visibly, and critics built on ImageNet-pretrained features can lower FID in the same way. Stein et al. (2023) compare seventeen scores with a large human study, find that Inception-based FID underrates the realism of diffusion models, and recommend DINOv2 features; recent image-generation papers report the Fréchet distance in DINOv2 features next to FID (Oquab et al. 2024; Karras, Aittala, Kynkäänniemi, et al. 2024). CMMD replaces both the Inception features and the Gaussian fit of FID: it computes the MMD of Section 16.2 with a Gaussian kernel on CLIP embed-

dings, which is unbiased and assumes nothing about the shape of the feature distribution (Jayasumana et al. 2024). Two further effects change the numbers without changing the images. Aliased resizing of the images before the feature network shifts FID by as much as the differences between published methods (Parmar et al. 2022). The finite-sample bias of FID depends on the generator as well as on n, so at small n it alone can reverse a ranking; extrapolating FID linearly in $1/n$ removes most of it (Chong and Forsyth 2020). A reader who trains a critic on pretrained features should evaluate with a different feature network.

Conditional scores. Section 16.6 measures fidelity, within-class diversity and alignment with three separate numbers. Two published scores combine them. The Fréchet joint distance fits a Gaussian to image and condition embeddings together, so a realistic image with the wrong condition changes the score (DeVries et al. 2019). The classification accuracy score trains a classifier on generated image–label pairs and tests it on real ones, so a collapsed or missing class shows up as errors on real images of that class (Ravuri and Vinyals 2019). Read them for evaluation protocols that a single number can report.

16.8.4 Conditions, Translation and Text

Modulation as a general layer. The modulation $a(c) \odot \phi + b(c)$ of Section 16.6 has a family of variants that differ in how the feature map is normalized first and where the scale and shift come from. Conditional batch normalization computes them from a question's embedding to steer an image network (Vries et al. 2017), and FiLM drops the normalization and makes the layer a general conditioning device (Perez et al. 2018). Adaptive instance normalization normalizes each image's channels separately and takes the scale and shift from the statistics of a style image (Huang and Belongie 2017); StyleGAN's style injection descends from it. SPADE makes the modulation spatial: a semantic layout supplies a separate scale and shift at each location, so the generator can paint a scene from a label map without normalization washing out the layout (Park et al. 2019).

Auxiliary classifiers. The auxiliary classifier critic of Odena et al. (2017) conditions the critic through a classification head instead of the projection head of Section 16.6, and trains the generator to make its samples receive the requested label. That loss rewards samples that are easy to classify, so it favors class prototypes over within-class diversity. Gong et al. (2019) remove the bias with a second classifier trained on generated pairs only, whose loss enters the generator's objective with the opposite sign. Read them for the alternative to projection and its repair.

Translation systems. Section 16.6 explains why pix2pix adds a critic to its L_1 loss. Without paired examples, CycleGAN gives up the conditional critic: each direction plays an unconditional game against its target collection, and a cycle-consistency loss asks that translating forward and back return the input. The cycle loss constrains the map without identifying it: composing a solution with any invertible map that preserves the target distribution gives another solution, and when the target carries less information than the source, the generator can hide the missing information in imperceptible signals (Chu et al. 2017). The two papers add the systems around these ideas (Isola et al. 2017; Zhu et al. 2017): an encoder–decoder generator with skip connections between matching resolutions (a U-Net), so that the source's layout reaches the output directly; and a patch critic that scores overlapping patches instead of the whole image, which is the local comparison the ambiguity argument asks for. They are the place to start for image-to-image applications.

Text as the condition. A caption is a sequence of tokens, not one class. GigaGAN feeds it to the generator through cross-attention from image features to the caption's tokens, as text-to-

image diffusion models do (Kang et al. 2023). By 2026 diffusion and autoregressive models dominate general text-to-image synthesis. GigaGAN is the reference for what a single-pass adversarial generator achieves at that task, and for the architectural changes it needed to scale.

16.8.5 Fixed Discrepancies and Learned Ones

The chapter's critics are learned. Could a fixed discrepancy, with no second network, serve as well? Generative moment matching networks minimize the fixed-kernel MMD of Section 16.2, with neither a critic nor an inner optimization (Li et al. 2015). IMLE pairs each data point with its nearest generated sample, which penalizes any data mode left without a nearby sample (Li and Malik 2018). Sliced Wasserstein generators average one-dimensional transport costs along random projections (Deshpande et al. 2018), and Sinkhorn divergences interpolate between optimal transport and MMD (Genevay et al. 2018).

On natural images, several of these became competitive only after a learned component was added. MMD-GAN learns the kernel adversarially (C.-L. Li et al. 2017), and sliced-Wasserstein generators add a discriminator to choose informative projections, because random directions carry little signal in high dimensions (Deshpande et al. 2018). This agrees with the estimation argument of Section 16.2: a discrepancy can separate distributions in principle and still be hard to estimate from samples. Sriperumbudur et al. (2012) give the convergence rates of the empirical estimators behind that argument, and Bińkowski et al. (2018) analyze the learned kernels of MMD-GAN and, from the same analysis, propose KID, the score of Section 16.5.

16.8.6 Adversarial Terms in Recent Systems

Section 16.7 explains why an adversarial term survives in one-step generators, autoencoders and vocoders. The systems below show the range of reasons; the numbers are those their papers reported between 2024 and 2026.

Tokenizers. Most latent image generators run on an autoencoder, and the adversarial term in that autoencoder decides how much texture the decoder can produce. VQGAN combined reconstruction and perceptual losses with a patch critic (Esser et al. 2021), and the Stable Diffusion autoencoder kept the combination (Rombach et al. 2022). The ViTok study measures both sides of the trade-off that Section 16.7 derives: fine-tuning its decoder adversarially lowered reconstruction FID to 0.50 at some cost in PSNR and SSIM, while the FID of images generated in its latent space improved only from 5.5 to 4.9 (Hansen-Estruch et al. 2025). Its successor ViTok-v2 scales the autoencoder to five billion parameters and replaces both the adversarial loss and the LPIPS perceptual loss with a fixed loss on self-supervised DINOv3 features (Hansen-Estruch et al. 2026). Read together, the two papers show what a critic adds to a tokenizer at moderate scale and what replaced it at large scale.

Distillation with real data. A distilled student that only matches its teacher can be no better than the teacher. DMD2 adds an adversarial loss against real images, which lets its ImageNet 64×64 student reach FID 1.28, below its teacher (Yin, Gharbi, Park, et al. 2024). LADD reuses the teacher diffusion model as the critic's backbone and finds that synthetic teacher samples train it well enough to drop the distillation loss, leaving only the adversarial objective (Sauer et al. 2024). The FLUX.1-schnell model card states that its twelve-billion-parameter rectified-flow transformer was trained this way to generate in one to four steps (Black Forest Labs 2024). Whole-image FID separates these few-step methods poorly. The

SDXL-Lightning report also computes FID on image patches, which responds to local texture, and finds larger differences there (Lin, Wang, et al. 2024): the ambiguity that the adversarial term addresses sits in fine detail, where a whole-image score averages it away.

Few steps without a critic. An adversarial term is not the only route to few-step generation. MeanFlow trains a one-step generator from scratch, with no teacher, distillation or critic, and reports FID 3.43 on ImageNet 256×256 (Geng et al. 2025). Continuous-time consistency models report FID 1.88 on ImageNet 512×512 with two network evaluations and no adversarial loss (Lu and Song 2025). Both replace the critic with a regression target designed to stay informative at low step counts.

Vocoders and video. RAF applies the pairing loss of Section 16.3 to waveforms; a BigVGAN-base trained with it, with twelve percent of the parameters, outperforms the full BigVGAN trained with the least-squares loss in perceptual quality (Lee and Choi 2026). WaveFM reaches single-step synthesis without a critic, by flow matching followed by consistency distillation (Luo et al. 2025). For video, adversarial post-training fine-tunes a diffusion generator against real data with a zero-centered gradient penalty from the family of Section 16.4, and produces two seconds of 1280×720 video at 24 frames per second in one forward evaluation (Lin et al. 2025).

Likelihood models as critics at scale. Direct discriminative optimization, the construction of Section 16.7 that uses the likelihood model itself as the critic, fine-tunes pretrained EDM diffusion models and lowers FID from 1.79 to 1.30 on CIFAR-10 and from 1.58 to 0.97 on ImageNet 64×64 (K. Zheng, Chen, Chen, et al. 2025).

Chapter 17

Diffusion Models

The generators of Chapter 16 learned to draw garments and sprites through a game between two networks, and that chapter closed with four limits the game leaves in place: no density, two networks and a tuned penalty, no cost for a missing mode, and instability at scale. It pointed instead to diffusion and flow models, trained by regression on a target computed from each training example. A classifier regresses onto a given label; a generator has nothing to copy, and its image must come out of noise. Regression onto what, then, and how can fitting a target produce something new?

The idea came from physics: in non-equilibrium thermodynamics, structure such as a drop of ink in water is destroyed by many small random steps, and a small step's reversal has the same simple form. Sohl-Dickstein et al. (2015) noised images in the same way and trained a network to undo each step; Ho et al. (2020) made the undoing a regression onto the added noise and matched the sample quality of adversarial training. Others reached the same model from the gradient of the log-density (Song and Ermon 2019) and from the velocity of a path between noise and data (Lipman et al. 2023; Liu et al. 2023; Albergo et al. 2023). All routes arrive at one network that predicts the clean point from a noisy one at every noise level, and a sampler that integrates its predictions.

What does a network that removes noise know about the data (Section 17.1)? How do we walk from noise to data, and which regression trains the walk (Section 17.2)? What changes for images, and how do we steer and judge them (Section 17.3)? Which sampling steps can we skip, and what does a sampler do when run backwards (Section 17.4)? Is the regression maximum likelihood, and which of its choices matter (Section 17.5)? Why do current systems diffuse a compressed code with a transformer (Section 17.6)? Why does a single step fail, and what fixes it (Section 17.7)? What is noise for a token (Section 17.8)?

Along the way the four limits of the game find answers: a density, one regression instead of two networks, a loss under which every missing mode costs, and a transformer trained stably at scale. The mathematics lives in Chapter 38, to which the chapter points instead of repeating it; image generation at scale, text-to-image and video belong to a later part of this book.

Resources and Further Reading

These resources accompany particular sections; Section 17.9 turns to the methods the chapter does not teach.

- *The Principles of Diffusion Models* (Lai et al. 2025) derives diffusion models along the variational, score-based and flow-based routes and shows where they meet.
- The notes, lectures and labs of MIT's course 6.S184 (Holderrieth and Erives 2025) teach flow matching, latent spaces with transformer backbones, and discrete diffusion, with labs that build each model from scratch.
- The *Flow Matching Guide and Code* (Lipman et al. 2024) and its library give the training loop of Section 17.2 in full, with proofs.
- *Step-by-Step Diffusion* (Nakkiran et al. 2025) builds the deterministic sampler up from a single point mass and derives the samplers of Section 17.4 without stochastic calculus.
- *Diffusion Meets Flow Matching* (Gao et al. 2025) shows with interactive figures that diffusion and flow samplers are one algorithm under a change of schedule, and that the weighting matters more than the schedule.
- Sander Dieleman's blog explains how noise acts on image frequencies (Dieleman 2024), why generators work in a latent space (Dieleman 2025), what flow maps learn (Dieleman 2026b), and how continuous and discrete diffusion compete for language (Dieleman 2026a).
- The official DDPM implementation of Ho et al. (2020) and the score SDE implementation of Y. Song, Sohl-Dickstein, et al. (2021) contain the full-scale networks and samplers that the small models of Section 17.3 imitate.
- The MDLM repository of Sahoo et al. (2024) trains the masked diffusion language model of Section 17.8.
- The *Deep Generative Models* course (Kuleshov 2023) and Yang Song's blog post (Song 2021) tell the score-based history behind Section 17.1.
- Lilian Weng's *What Are Diffusion Models?* (Weng 2021) and the tutorial of Luo (2022) work through the variational algebra that Section 17.5 compresses.

17.1 Denoising and the Score

The previous chapter ended with a fact about squared error: a network trained by squared error learns the conditional mean of its target given its input, and where the input leaves the target ambiguous, that mean averages the possibilities. A denoiser is such a network: shown a noisy photograph, it returns the average of all clean photographs that could have produced it. For the generators of Chapter 16 that average was a defect. Here it is the starting point, because diffusion and flow models are trained by this same regression: add noise to a training example and ask the network for the clean example back.

What does a denoiser know about the data it was trained on? The noise filter of a camera app, trained on millions of photographs, restores edges and faces in a grainy picture. Fed pure static, it does not return static: it sharpens whatever faint structure it finds into the kinds of shapes that photographs contain. Statisticians answered the question long before deep learning. Herbert Robbins and Koichi Miyasawa showed, and Efron (2011) popularized as *Tweedie's formula*, that the best denoiser moves the noisy point along the gradient of the log-density of the noisy data, a gradient called the *score*. A denoiser and a score are therefore one object, and Vincent (2011) showed that training a denoiser by regression fits the score.

This matters because a density needs a normalizer, an integral over all possible inputs that

nobody can compute for images, while a score needs none and is enough to draw samples. On a small distribution where everything is exact, we show that a denoiser is a posterior mean and measure what one noise level can and cannot do; then we denoise images.

```
torch.set_float32_matmul_precision('high')  # tf32 matmuls on tensor cores
```

17.1.1 The Running Example

We want one distribution on which every quantity of the chapter is exact at every noise level, so that each learned field and sampler can be checked against the truth. A mixture of Gaussians in the plane qualifies. Ours has three components with means $(-2.5, -1.5)$, $(2.5, -1.5)$ and $(0, 2.5)$, mixture weights $\pi_k = 0.5, 0.3, 0.2$, and a common standard deviation $s = 0.5$ per coordinate. The modes are far apart compared with their width, so the density has deep valleys between them, and the unequal weights give a sampler something to get wrong.

The *score* of a density p is the gradient of its logarithm, $\nabla \log p(\mathbf{x})$, a vector field that points in the direction in which the density grows fastest. For the mixture $p(\mathbf{x}) = \sum_k \pi_k \, \mathcal{N}(\mathbf{x}; \mu_k, s^2\mathbf{I})$ it reads

$$\nabla \log p(\mathbf{x}) = \sum_k r_k(\mathbf{x}) \, \frac{\mu_k - \mathbf{x}}{s^2}, \qquad r_k(\mathbf{x}) = \frac{\pi_k \, \mathcal{N}(\mathbf{x}; \mu_k, s^2\mathbf{I})}{p(\mathbf{x})}. \tag{17.1}$$

Each component pulls the point toward its own mean, and the pulls are averaged with the *responsibilities* $r_k(\mathbf{x})$, the posterior probabilities that $\mathbf{x}$ was drawn from component k (Figure 17.1). The weights π_k enter only through the responsibilities. Near a mode one responsibility is close to one whatever the weights are, so the score there is that of a single Gaussian. The weights change the score only in the valleys, where few points lie.

Adding Gaussian noise of standard deviation σ to a draw adds σ^2 to the variance of each component, so the noised distribution is again a mixture, with s^2 replaced by $s^2 + \sigma^2$. `GaussianMixture` holds this mixture, with its density, score and posterior mean in closed form at every noise level $\mathbf{x}_t = \alpha \mathbf{x}_0 + \sigma\epsilon$, and the velocity of the linear path of Section 17.2; in this section $\alpha = 1$.

```
class GaussianMixture:
    """A mixture of isotropic Gaussians, exact at every noise level."""
    def __init__(self, means, weights, std, device=None):
        self.means = torch.tensor(means, dtype=torch.float32, device=device)
        self.weights = torch.tensor(weights, dtype=torch.float32,
                                    device=device)
        self.std = std

    @staticmethod
    def three_modes(device=None):
        """The chapter's running example, with mode weights 0.5/0.3/0.2."""
        return GaussianMixture([[-2.5, -1.5], [2.5, -1.5], [0.0, 2.5]],
                               [0.5, 0.3, 0.2], 0.5, device)

    def sample(self, n):
```

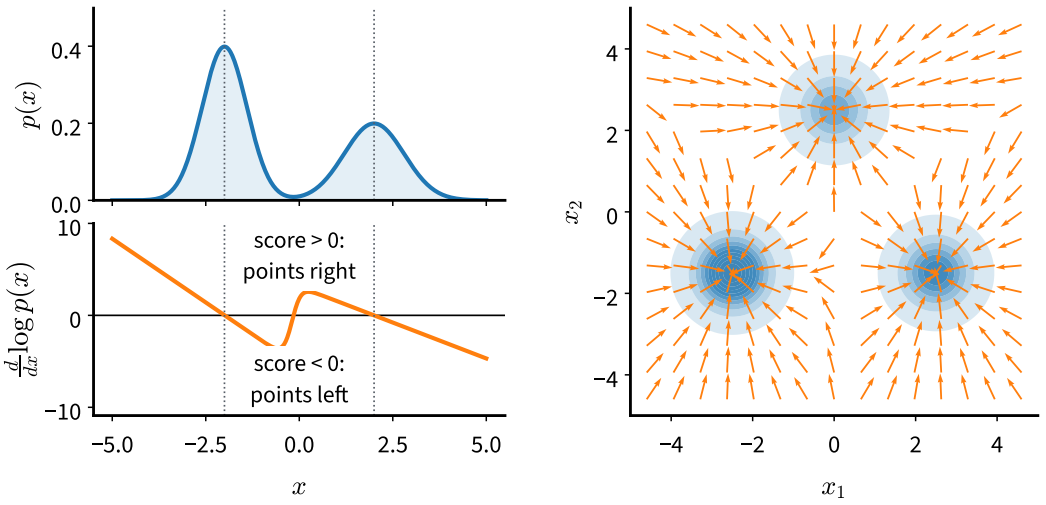


Figure 17.1: The score is the gradient of the log-density. Left: a bimodal density in one dimension (top) and its score (bottom), zero at both modes and at the valley between them. Right: the running example, with its density as shaded contours and its score field as arrows that point toward the modes (arrow lengths clipped).

```
        """n draws from the mixture."""
        k = torch.multinomial(self.weights, n, replacement=True)
        return self.means[k] + self.std * torch.randn(
            n, self.means.shape[1], device=self.means.device)

    def _modes(self, x, alpha, sigma):
        """Per-mode log densities, offsets x - alpha mu_k, variances, alpha."""
        a, s = (torch.as_tensor(v, dtype=x.dtype, device=x.device)
                .reshape(-1, 1, 1) for v in (alpha, sigma))
        var = (a * self.std) ** 2 + s ** 2
        r = x[:, None] - a * self.means
        log_p = (torch.log(self.weights) - (r ** 2).sum(-1) / (2 * var[..., 0])
                 - x.shape[1] / 2 * torch.log(2 * torch.pi * var[..., 0]))
        return log_p, r, var, a

    def log_prob(self, x, alpha=1.0, sigma=0.0):
        """log p(x) for x = alpha x0 + sigma eps."""
        return torch.logsumexp(self._modes(x, alpha, sigma)[0], 1)

    def score(self, x, alpha=1.0, sigma=0.0):
        """The gradient of log p(x) for x = alpha x0 + sigma eps."""
        log_p, r, var, _ = self._modes(x, alpha, sigma)
        return -(torch.softmax(log_p, 1)[..., None] * r / var).sum(1)

    def posterior_mean(self, x, alpha, sigma):
        """E[x0 | x]; finite at alpha = 0 and at sigma = 0."""
        log_p, r, var, a = self._modes(x, alpha, sigma)
        return (torch.softmax(log_p, 1)[..., None]
```

```
                * (self.means + a * self.std ** 2 / var * r)).sum(1)

    def velocity(self, x, t):
        """E[eps - x0 | x_t] on the linear path x_t = (1 - t) x0 + t eps."""
        log_p, r, var, a = self._modes(x, 1 - t, t)
        eps = (1 - a) * r / var                        # E[eps | x_t, mode k]
        x0 = self.means + a * self.std ** 2 / var * r  # E[x0 | x_t, mode k]
        return (torch.softmax(log_p, 1)[..., None] * (eps - x0)).sum(1)
```

Two kinds of statistic score a set of points against the mixture: the mean negative log-likelihood under the true density, and a `summary` of three numbers, the mode weights (each point counted at its nearest mean), the fraction of strays beyond three standard deviations from every mean, and the mean squared distance of the rest to their mean.

```
@d2l.add_to_class(GaussianMixture)
def nll(self, x):
    """Mean negative log-likelihood of clean points, in nats."""
    return -self.log_prob(x).mean()

@d2l.add_to_class(GaussianMixture)
def summary(self, x):
    """Mode weights by nearest mean, stray fraction, spread of the rest."""
    d2 = ((x[:, None] - self.means) ** 2).sum(-1)
    near, k = d2.min(1)
    inside = near < (3 * self.std) ** 2
    weights = torch.bincount(k, minlength=len(self.weights)) / len(x)
    return dict(weights=[round(w, 3) for w in weights.tolist()],
                stray=round(1 - inside.float().mean().item(), 4),
                spread=round(near[inside].mean().item(), 3))
```

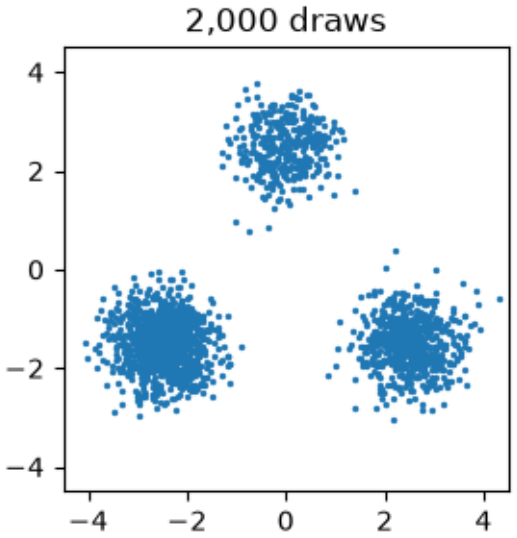
`scatter_panels` draws point sets side by side on shared, equal axes, with the training data in grey behind them.

```
def scatter_panels(samples, titles=None, data=None, lim=4.5,
                   size=3.0):
    """Scatter 2-D point sets side by side on equal, shared axes."""
    _, axes = d2l.plt.subplots(1, len(samples), squeeze=False,
                               figsize=(size * len(samples), size))
    for i, (ax, pts) in enumerate(zip(axes[0], samples)):
        if data is not None:
            ax.scatter(*d2l.numpy(data).T, s=2, c='lightgray')
        ax.scatter(*d2l.numpy(pts).T, s=2)
        ax.set(xlim=(-lim, lim), ylim=(-lim, lim), aspect='equal')
        if titles:
            ax.set_title(titles[i])
    return axes[0]
```

We draw 2,000 points, the training set of this section's two-dimensional experiments, and score them.

```
torch.manual_seed(0)
mix = d2l.GaussianMixture.three_modes()
data = mix.sample(2000)
print(mix.summary(data), f'NLL {mix.nll(data):.3f}')
d2l.scatter_panels([data], ['2,000 draws']);
```

```
{'weights': [0.5, 0.317, 0.183], 'stray': 0.012, 'spread': 0.488} NLL 2.504
```



The weights come out near 0.5, 0.3 and 0.2, about one percent of the points stray, and the spread is just under $2s^2 = 0.5$, because the test cuts off the outer tail of each component. A perfect sampler would print the same values, and no model can beat the negative log-likelihood of about 2.5 nats per point on average.

17.1.2 The Optimal Denoiser

What does a denoiser learn, how is that a score, and why is the score the object to learn?

17.1.2.1 Noise and the Regression Target

Pick a noise level $\sigma > 0$, draw a clean point $\mathbf{x}_0 \sim p$ and standard Gaussian noise $\epsilon \sim \mathcal{N}(\mathbf{0}, \mathbf{I})$, and form the noisy point $\tilde{\mathbf{x}} = \mathbf{x}_0 + \sigma\epsilon$. Its density p_σ is the data density blurred by a Gaussian of width σ, positive and smooth everywhere even when p is not. A denoiser D is trained to minimize $\mathbb{E}\|D(\tilde{\mathbf{x}}) - \mathbf{x}_0\|^2$, and squared error is minimized by the conditional mean (Section 38.5.1.2), so the optimal denoiser is the *posterior mean* $\mathbb{E}[\mathbf{x}_0 \mid \tilde{\mathbf{x}}]$, the average of the clean points that could have produced $\tilde{\mathbf{x}}$, each weighted by how likely it is to have done so.

The same regression can aim at a different target. Given the clean point, the noisy point is Gaussian around it, and the score of this conditional density is

$$\nabla_{\tilde{\mathbf{x}}} \log p_\sigma(\tilde{\mathbf{x}} \mid \mathbf{x}_0) = \frac{\mathbf{x}_0 - \tilde{\mathbf{x}}}{\sigma^2} = -\frac{\epsilon}{\sigma}, \tag{17.2}$$

a vector pointing from the noisy point back to the clean one. *Denoising score matching* regresses a vector field $\mathbf{s}_\theta$ onto it (Vincent 2011):

$$J_{\text{DSM}}(\theta) = \tfrac{1}{2}\,\mathbb{E}_{\mathbf{x}_0,\epsilon}\left\|\mathbf{s}_\theta(\mathbf{x}_0 + \sigma\epsilon) + \frac{\epsilon}{\sigma}\right\|^2. \tag{17.3}$$

The minimizer is again a conditional mean, now of the conditional score, and Vincent's theorem identifies it: *the score of the noised data is the posterior mean of the conditional score*,

$$\mathbb{E}\Big[-\frac{\epsilon}{\sigma}\,\Big|\,\tilde{\mathbf{x}}\Big] = \nabla \log p_\sigma(\tilde{\mathbf{x}}). \tag{17.4}$$

The regression on per-pair targets therefore fits the score of p_σ, which we never wrote down; Section 38.5.1.2 proves the theorem in two lines of Bayes' rule.

`TimeMLP` is the network of every two-dimensional experiment: an MLP on the point plus sinusoidal features of a noise level or time from `time_embedding`, which the image networks share. Here the noise level is fixed.

```
def time_embedding(t, dim):
    """Sines and cosines of t at dim / 2 geometric frequencies."""
    freqs = torch.exp(-math.log(10000) / (dim // 2)
                      * torch.arange(dim // 2, device=t.device))
    args = t.float()[:, None] * freqs
    return torch.cat([args.sin(), args.cos()], 1)

class TimeMLP(nn.Module):
    """An MLP on a point x and the time features of one or more times."""
    def __init__(self, hidden=256, depth=3, *, dim=2, num_times=1,
                 time_dim=64):
        super().__init__()
        self.time_dim = time_dim
        self.embed = d2l.make_mlp([time_dim * num_times, hidden, hidden],
                                  nn.SiLU())
        self.inp = nn.Linear(dim, hidden)
        self.body = nn.Sequential(  # depth layers, each after a SiLU
            nn.SiLU(), d2l.make_mlp([hidden] * depth + [dim], nn.SiLU()))

    def forward(self, x, *ts):
        feats = [time_embedding(t, self.time_dim) for t in ts]
        return self.body(self.inp(x) + self.embed(torch.cat(feats, 1)))
```

`train_regression` is the one training loop of the chapter: Adam on a loss of the model and a fresh batch, constant or cosine-decayed learning rate, and `d2l.EMA` from Section 16.5 on the weights, whose half-life h is the number of updates after which a weight's contribution to the average halves. On a GPU each version compiles its training step (Section 13.3): the networks of this chapter are small enough that launching their kernels one at a time would cost more than running them.

```
def train_regression(model, loss_fn, batch_fn, num_steps, *,
                     lr=1e-3, cosine=False, ema_half_life=1000,
```

```
                    compiled=True):
    """Adam on loss_fn(model, batch_fn()) -> (model, EMA model, losses)."""
    params = list(model.parameters())  # fused Adam: one kernel per step on a GPU
    opt = torch.optim.Adam(params, lr=lr, fused=params[0].is_cuda)
    sched = (torch.optim.lr_scheduler.CosineAnnealingLR(opt, num_steps)
             if cosine else None)
    if compiled and params[0].is_cuda:  # loss and gradient as CUDA graphs
        loss_fn = torch.compile(loss_fn, mode='reduce-overhead',
                                dynamic=False)
    ema, losses = d2l.EMA(model, ema_half_life), []
    for _ in range(num_steps):
        loss = loss_fn(model, batch_fn())
        opt.zero_grad()
        loss.backward()
        opt.step()
        if sched:
            sched.step()
        ema.update(model)
        losses.append(loss.detach().clone())  # the graph reuses its output
    return model, ema.model, torch.stack(losses)
```

`fit_score` trains a `TimeMLP` with two hidden layers of 128 units on Equation 17.3, each step on 256 of the 2,000 training points with fresh noise, and returns the score as a function of the point alone.

```
def dsm_loss(net, x, sigma):
    """Denoising score matching at one noise level: regress onto -eps/sigma."""
    eps = torch.randn_like(x)
    s = net(x + sigma * eps, torch.full((len(x),), sigma))
    return 0.5 * ((s + eps / sigma) ** 2).sum(1).mean()

def fit_score(data, sigma, seed=1, num_steps=2000):
    """Train a score network at one noise level and return it as a function."""
    torch.manual_seed(seed)
    batch_fn = lambda: data[torch.randint(0, len(data), (256,))]
    net, _, _ = d2l.train_regression(
        d2l.TimeMLP(128, 2), lambda net, x: dsm_loss(net, x, sigma),
        batch_fn, num_steps)
    return lambda x: net(x, torch.full((len(x),), sigma)).detach()
```

17.1.2.2 Tweedie's Formula

The two regressions above, onto the clean point and onto $-\epsilon/\sigma$, have targets related by the map $\mathbf{x}_0 \mapsto (\mathbf{x}_0 - \tilde{\mathbf{x}})/\sigma^2$, which is affine once $\tilde{\mathbf{x}}$ is given, so their minimizers, two conditional means, are related by the same map. Solving for the posterior mean gives *Tweedie's formula* (Efron 2011),

$$\mathbb{E}[\mathbf{x}_0 \mid \tilde{\mathbf{x}}] = \tilde{\mathbf{x}} + \sigma^2 \nabla \log p_\sigma(\tilde{\mathbf{x}}). \tag{17.5}$$

The best denoiser takes one step of size σ^2 up the score of the noisy density (Figure 38.22). A denoiser, a score and a noise predictor are therefore three scalings of one posterior mean:

$$\hat{\mathbf{x}}_0(\tilde{\mathbf{x}}) = \tilde{\mathbf{x}} + \sigma^2\,\hat{\mathbf{s}}(\tilde{\mathbf{x}}), \qquad \hat{\epsilon}(\tilde{\mathbf{x}}) = \frac{\tilde{\mathbf{x}} - \hat{\mathbf{x}}_0(\tilde{\mathbf{x}})}{\sigma} = -\sigma\,\hat{\mathbf{s}}(\tilde{\mathbf{x}}). \tag{17.6}$$

A network trained to output any one of the three gives the other two.

The smallest case shows what the step does. Let the data be Gaussian, $\mathcal{N}(\mu, v\mathbf{I})$. Then $p_\sigma = \mathcal{N}(\mu, (v+\sigma^2)\mathbf{I})$ has the score $-(\tilde{\mathbf{x}} - \mu)/(v+\sigma^2)$, and Tweedie's formula gives

$$\mathbb{E}[\mathbf{x}_0 \mid \tilde{\mathbf{x}}] = \mu + \frac{v}{v+\sigma^2}\,(\tilde{\mathbf{x}} - \mu).$$

The denoiser shrinks the noisy offset from the mean by the factor $v/(v+\sigma^2)$, and the posterior variance per coordinate is $v\sigma^2/(v+\sigma^2)$. With $v = s^2 = 0.25$ and $\sigma = 0.3$ the factor is 0.74, and the squared error per coordinate falls from $\sigma^2 = 0.09$ to 0.066. Away from the valleys the posterior mean of the running example is this shrinkage toward the nearest mean, so over the two coordinates of a point denoising should lower the squared error from 0.18 to about 0.13.

The cell checks the formula at $\sigma = 0.3$: with the exact score of p_σ, Tweedie's step should reproduce the exact posterior mean, and with the network from `fit_score` it gives a learned denoiser to compare with the best one on 20,000 fresh noisy points.

```
sigma = 0.3
score = fit_score(data, sigma)
x0 = mix.sample(20000)
xt = x0 + sigma * torch.randn_like(x0)
exact = mix.posterior_mean(xt, 1.0, sigma)
gap = xt + sigma ** 2 * mix.score(xt, 1.0, sigma) - exact
print(f'Tweedie with the exact score: largest gap {gap.abs().max():.1e}')
for name, x0_hat in (('noisy point', xt), ('posterior mean', exact),
                     ('Tweedie, trained score', xt + sigma ** 2 * score(xt))):
    print(f'{name}: squared error {((x0_hat - x0) ** 2).sum(1).mean():.4f}')
```

```
Tweedie with the exact score: largest gap 4.8e-07
noisy point: squared error 0.1782
posterior mean: squared error 0.1317
Tweedie, trained score: squared error 0.1329
```

With the exact score the two sides agree to single-precision rounding. The exact posterior mean lowers the squared error from 0.18 to about 0.13, as the Gaussian case predicted, and the denoiser built from the trained score comes within about one percent of it: after two thousand steps the network is a nearly optimal denoiser, and its score a usable score.

17.1.2.3 The Normalizer Never Appears

Why not fit a density and differentiate it? The most flexible way to let a network define a density is an *energy-based model*,

$$p_\theta(\mathbf{x}) = \frac{\exp(-E_\theta(\mathbf{x}))}{Z_\theta}, \qquad Z_\theta = \int \exp(-E_\theta(\mathbf{x}))\,d\mathbf{x}. \tag{17.7}$$

Any network with a scalar output can serve as the *energy* E_θ, and low energy means high density (Figure 17.2). The normalizer makes learning meaningful: because the total mass is fixed, raising the density on the training points lowers it elsewhere. But Z_θ is an integral over the whole input space that changes with every update, and the gradient of the log-likelihood contains $\nabla_\theta \log Z_\theta = -\mathbb{E}_{p_\theta}[\nabla_\theta E_\theta]$, an expectation under the model itself, which only a Markov chain can supply.

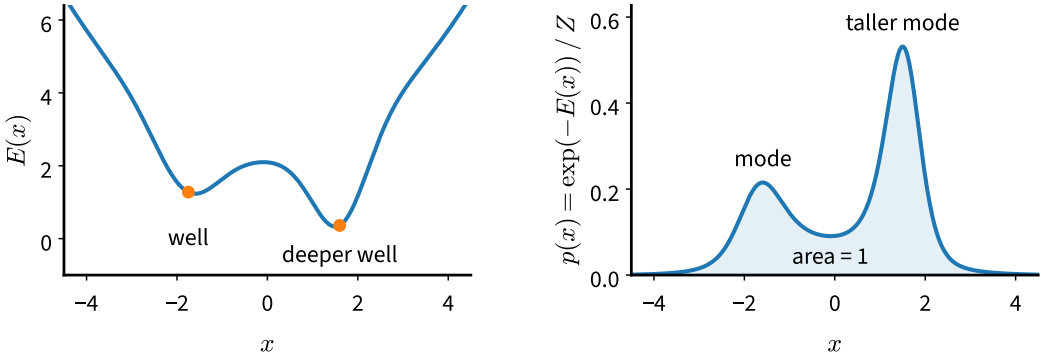


Figure 17.2: An energy and its density in one dimension. Left: the energy $E(x)$, a smooth function with a confining quadratic term. Right: the density $\exp(-E(x))/Z$. Wells of the energy become modes of the density.

Estimating Z directly fails in high dimensions. Importance sampling (Section 36.8.2) draws points from a proposal q and averages the weights $w = e^{-E}/q$, whose mean is Z. Take the energy $E(\mathbf{x}) = \|\mathbf{x} - \mathbf{m}\|^2/2$ with $\mathbf{m}$ the all-ones vector in d dimensions, and the proposal $q = \mathcal{N}(\mathbf{0}, \mathbf{I})$, only one standard deviation off in each coordinate. The normalized weight $w/Z = \exp(\mathbf{m}^\top \mathbf{x} - d/2)$ is log-normal with mean one and $\mathbb{E}[(w/Z)^2] = e^d$, so the relative error of an average of n weights is $\sqrt{(e^d - 1)/n}$. The cell compares this prediction with the estimate and with the *effective sample size* $(\sum_i w_i)^2 / \sum_i w_i^2$, the number of draws that carry appreciable weight.

```
torch.manual_seed(1)
n = 100_000
print(f'{"d":>3} {"estimate / Z":>13} {"predicted rel. error":>21} {"ESS":>8}')
for d in (1, 2, 4, 8, 16, 32):
    x = torch.randn(n, d)                       # proposal q = N(0, I)
    log_w = x.sum(1) - d / 2                    # log(w / Z), m = (1, ..., 1)
    log_sum = torch.logsumexp(log_w, 0)
    estimate = torch.exp(log_sum - math.log(n))
    ess = torch.exp(2 * log_sum - torch.logsumexp(2 * log_w, 0))
    print(f'{d:>3} {estimate:>13.3f} {math.sqrt((math.exp(d) - 1) / n):>21.3g}'
          f' {ess:>8.1f}')
```

```
  d  estimate / Z  predicted rel. error      ESS
  1         0.999                0.00415  37155.5
  2         1.017                0.00799  12501.1
  4         1.009                 0.0232   2620.7
  8         1.038                  0.173    132.1
 16         0.529                   9.43     72.3
```

```
32          0.081              2.81e+04     11.0
```

Up to $d = 8$ the estimate is close to one. At $d = 16$ the predicted relative error is about nine and only tens of draws carry weight, so a run that lands near the truth does so by luck. At $d = 32$ the estimate falls several times to an order of magnitude short: the weights that would raise the average to Z belong to draws too rare to appear among 10^5. An image has hundreds of thousands of coordinates.

The score avoids all of this. The normalizer does not depend on $\mathbf{x}$, so it vanishes from the gradient in $\mathbf{x}$:

$$\nabla_{\mathbf{x}} \log p_\theta(\mathbf{x}) = -\nabla_{\mathbf{x}} E_\theta(\mathbf{x}). \tag{17.8}$$

To compare a field $\mathbf{s}$ with the data we use the *Fisher divergence*, the mean squared distance between the two fields where the data lie,

$$D_{\mathrm{F}}(p, \mathbf{s}) = \tfrac{1}{2}\, \mathbb{E}_{\mathbf{x}\sim p} \big\| \mathbf{s}(\mathbf{x}) - \nabla \log p(\mathbf{x}) \big\|^2 . \tag{17.9}$$

It contains the unknown data score, but integration by parts moves the derivative onto the model. *Hyvärinen's identity* (Hyvärinen 2005) states that for every smooth field that decays fast enough,

$$D_{\mathrm{F}}(p, \mathbf{s}) = \mathbb{E}_{\mathbf{x}\sim p} \big[\tfrac{1}{2} \|\mathbf{s}(\mathbf{x})\|^2 + \nabla \cdot \mathbf{s}(\mathbf{x}) \big] + C, \tag{17.10}$$

with a constant C that does not depend on $\mathbf{s}$ (Section 38.5.1). The right side is an average over data, with no normalizer and no chain, but its divergence term, the trace of a Jacobian, costs one backward pass per input dimension. Denoising removes that cost too: by Vincent's theorem, Equation 17.3 is the Fisher divergence to p_σ up to a constant.

17.1.3 Limits of a Single Noise Level

A denoiser trained at one noise level gives the score of one blurred distribution. How good is that score, what can it not see, and what does a sampler driven by it produce?

17.1.3.1 The Loss Floor and the Noise Level

The denoising loss does not reach zero, because even the best field leaves the conditional variance of the target: the minimum of $2J_{\mathrm{DSM}}$ is the average of $\operatorname{tr} \operatorname{Cov}(\epsilon/\sigma \mid \tilde{\mathbf{x}})$, which is $d\,(1/\sigma^2 - 1/(v + \sigma^2))$ in the Gaussian case and in general grows like d/σ^2 as σ shrinks (Section 38.5.1.2). A training loss that levels off above zero is therefore expected. At a small noise level the target $-\epsilon/\sigma$ is large and mostly noise, and a network trained on a finite sample fits it poorly.

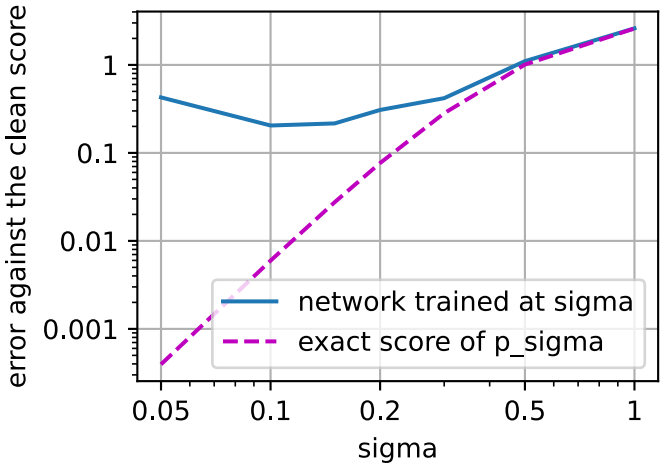
A large noise level has the opposite problem. The regression is well conditioned, but its answer is the score of p_σ, not of p. For one Gaussian component $\mathcal{N}(\mu, s^2\mathbf{I})$ in d dimensions the two scores are $-(\mathbf{x} - \mu)/(s^2 + \sigma^2)$ and $-(\mathbf{x} - \mu)/s^2$, and averaging the squared norm of their difference over p gives the bias

$$\tfrac{1}{2}\, \mathbb{E}_p \big\| \nabla \log p_\sigma - \nabla \log p \big\|^2 = \frac{d\, \sigma^4}{2 s^2 (s^2 + \sigma^2)^2} .$$

The bias grows like σ^4 at first. For $s = 0.5$ and $d = 2$ it is 0.0004 at $\sigma = 0.05$, 0.28 at $\sigma = 0.3$ and 2.6 at $\sigma = 1$, and the well-separated mixture has nearly the same values. The cell trains a network at each of seven noise levels and measures its error against the clean score on 20,000 test points, next to the bias of the exact smoothed score.

```
sigmas = [0.05, 0.1, 0.15, 0.2, 0.3, 0.5, 1.0]
scores = {s: fit_score(data, s) for s in sigmas}
x_test = mix.sample(20000)
def score_error(score, x, target):
    return 0.5 * ((score(x) - target) ** 2).sum(1).mean().item()
learned = [score_error(scores[s], x_test, mix.score(x_test)) for s in sigmas]
bias = [score_error(lambda x: mix.score(x, 1.0, s), x_test, mix.score(x_test))
        for s in sigmas]
print('trained at sigma:', [round(e, 3) for e in learned])
print('exact smoothed:  ', [round(e, 4) for e in bias])
d2l.plot(sigmas, [learned, bias], 'sigma', 'error against the clean score',
         legend=['network trained at sigma', 'exact score of p_sigma'],
         xscale='log', yscale='log')
```

```
trained at sigma: [0.427, 0.205, 0.216, 0.308, 0.419, 1.103, 2.604]
exact smoothed:   [0.0004, 0.006, 0.0275, 0.0767, 0.2826, 1.0084, 2.5955]
```



The learned error traces a U. To the right it follows the bias line, within about 0.1 from $\sigma = 0.5$ on: the networks fit their blurred targets well, and the targets are wrong. To the left it rises again although the bias vanishes, because the targets are mostly noise; retraining with another seed moves these points by up to a factor of two. The best single level lies between $\sigma = 0.1$ and 0.2, at an error of about 0.15 to 0.2.

17.1.3.2 Blindness to Mode Weights

An error of 0.2 in the Fisher divergence sounds small, but small compared with what? Compare the running example with its equal-weight copy. A model with these wrong weights loses 0.07 nats of likelihood per point, their Kullback–Leibler divergence. Its score, however, differs from the true one only through the responsibilities in Equation 17.1, which differ only in the valleys, where p is nearly zero, so the Fisher divergence, an average under

p, barely sees the difference.

The cell computes both divergences by quadrature on a 721×721 grid, because the integrand lives in the valleys, where few Monte Carlo draws land, and then repeats the comparison with both mixtures blurred by the same noise.

```
mix_eq = d2l.GaussianMixture(mix.means.tolist(), [1 / 3] * 3, mix.std)
g = torch.linspace(-9, 9, 721)
grid = torch.stack(torch.meshgrid(g, g, indexing='ij'), -1).reshape(-1, 2)
area = (g[1] - g[0]) ** 2                    # of one grid cell
for sigma in (0.0, 0.5, 1.0):
    log_p = mix.log_prob(grid, 1.0, sigma)
    diff = mix.score(grid, 1.0, sigma) - mix_eq.score(grid, 1.0, sigma)
    fisher = area * (log_p.exp() * 0.5 * (diff ** 2).sum(1)).sum()
    kl = area * (log_p.exp()
                 * (log_p - mix_eq.log_prob(grid, 1.0, sigma))).sum()
    print(f'sigma = {sigma}: Fisher divergence {fisher:.1e}, KL {kl:.3f} nats')
```

```
sigma = 0.0: Fisher divergence 1.5e-05, KL 0.069 nats
sigma = 0.5: Fisher divergence 1.4e-03, KL 0.069 nats
sigma = 1.0: Fisher divergence 9.5e-03, KL 0.064 nats
```

Between the clean mixtures the Fisher divergence is about 10^{-5}, far below the errors of the trained networks, none of which fell below 0.1, while the KL divergence is 0.07 nats. A score model trained near the clean data cannot learn the weights: a field with the wrong weights fits as well as one with the right weights, far within any network's error (Section 37.5.1, 3). Noise makes the weights visible. As σ grows, the components spread into the valleys (Figure 17.3), the responsibilities differ where p_σ has mass, and the Fisher divergence rises to about 10^{-3} at $\sigma = 0.5$ and 10^{-2} at $\sigma = 1$, while the KL divergence barely changes.

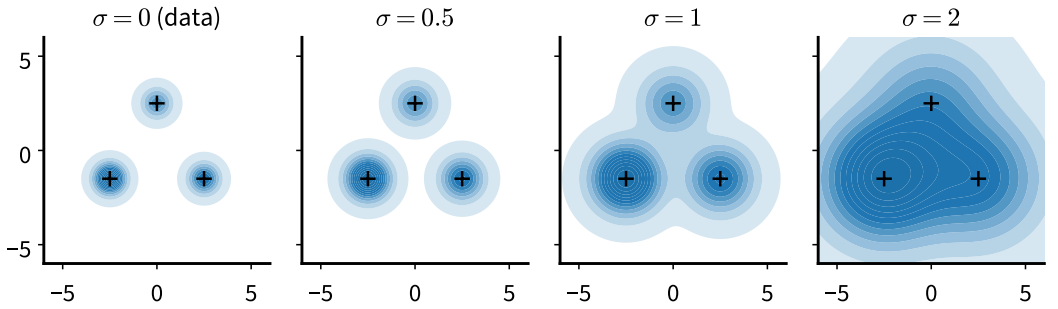


Figure 17.3: The running example blurred by Gaussian noise of growing standard deviation σ; crosses mark the component means. As σ grows, the valleys between the modes fill in, until at $\sigma = 2$ a single broad bump remains.

17.1.3.3 Sampling with a Score

A score is enough to draw samples. *Langevin dynamics* starts from any point and repeats

$$\mathbf{x} \leftarrow \mathbf{x} + \frac{h}{2}\,\mathbf{s}(\mathbf{x}) + \sqrt{h}\,\xi, \qquad \xi \sim \mathcal{N}(\mathbf{0}, \mathbf{I}), \tag{17.11}$$

with a small step size h. The drift moves the point up the log-density, and the noise keeps it from collapsing onto a mode. In the continuous-time limit, p is the stationary distribution whenever $\mathbf{s} = \nabla \log p$ (Section 38.5.2), and a chain run long enough forgets its start. The discrete update is slightly off: for a Gaussian of variance v its stationary variance is $v/(1 - h/(4v))$, a five percent excess for our components at $h = 0.05$.

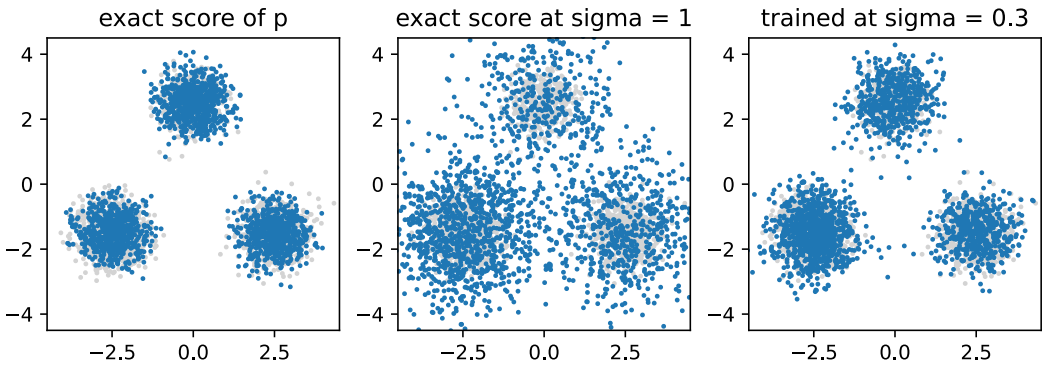
`langevin` runs this update with any score function.

```
@torch.no_grad()
def langevin(score_fn, x, step, num_steps):
    """Langevin steps x <- x + step / 2 * score + sqrt(step) * noise."""
    for _ in range(num_steps):
        x = x + step / 2 * score_fn(x) + step ** 0.5 * torch.randn_like(x)
    return x
```

We run three chains of 2,000 steps at $h = 0.05$ from the same 2,000 starting points, drawn from $\mathcal{N}(\mathbf{0}, 9\mathbf{I})$ so that each mode receives about a third of them: with the exact score of p, with the exact score of p_σ at $\sigma = 1$, and with the network trained at $\sigma = 0.3$, used as if it were the score of p.

```
torch.manual_seed(2)
x_init = 3 * torch.randn(2000, 2)
fields = {'exact score of p': mix.score,
          'exact score at sigma = 1': lambda x: mix.score(x, 1, 1),
          'trained at sigma = 0.3': scores[0.3]}
samples = [d2l.langevin(f, x_init, 0.05, 2000) for f in fields.values()]
print('start:', mix.summary(x_init))
for name, x in zip(fields, samples):
    print(f'{name}:', mix.summary(x))
d2l.scatter_panels(samples, list(fields), data=data);
```

```
start: {'weights': [0.308, 0.329, 0.363], 'stray': 0.751, 'spread': 1.051}
exact score of p: {'weights': [0.312, 0.335, 0.354], 'stray': 0.0095, 'spread': 0.501}
exact score at sigma = 1: {'weights': [0.498, 0.298, 0.204], 'stray': 0.4245, 'spread': 0.986}
trained at sigma = 0.3: {'weights': [0.486, 0.244, 0.27], 'stray': 0.063, 'spread': 0.649}
```



The exact score of p sends every chain to a mode with the right spread, but the weights stay those of the start, about a third each, because the chains do not cross the valleys: for the two lower modes the valley lies about 12 nats above the modes in $-\log p$, and the expected

crossing time is of the order of a million steps at this step size (9). The sampler is right within each mode and wrong about the weights, the same weights to which the Fisher divergence is blind.

The exact score of p_σ at $\sigma = 1$ crosses freely, because the noise has filled the valleys, and its weights land within about 0.02 of 0.5, 0.3 and 0.2. But its samples have twice the spread, and about 40% of them fail the three-standard-deviation test: they are samples of p_σ.

The trained network does neither job reliably. Its weights depend on the seed, staying near the start's in some runs and drifting in others, because nothing in a field fitted near the data fixes them. Several percent of its samples stray, against one to two percent for the exact score, because the field is wrong in the valleys where no training pair landed. Langevin steps survive in modern samplers only as a correction applied at each of many noise levels.

17.1.4 Denoising Images

Nothing in the denoising loss depends on the dimension of the data, so it applies to images unchanged. We use Fashion-MNIST (Section 3.2) with pixel values scaled to $[-1, 1]$ and noise of standard deviation $\sigma = 0.5$. The network is four 3×3 convolutions with 32 channels and SiLU activations, trained for 600 steps with minibatches of 128 images. It predicts the noise ϵ, so by Equation 17.6 its output divided by $-\sigma$ is a score of the noisy images, a field in 784 dimensions without any density, and $\tilde{\mathbf{x}} - \sigma\hat{\epsilon}$ is the denoised image.

`load_fashion_mnist_pm1` returns the Fashion-MNIST images scaled to $[-1, 1]$ (or as binary pixels) with their labels, and `show_gray` displays images as a grid.

```
def load_fashion_mnist_pm1(binary=False):
    """Fashion-MNIST (X, y, X_test, y_test), images in [-1, 1] or binary."""
    data, device = d2l.FashionMNIST(), d2l.try_gpu()  # on the GPU if any
    def prep(X):  # (n, 1, 28, 28) floats or (n, 28, 28) binary pixels
        return (X > 127).long() if binary else X[:, None].float() / 127.5 - 1
    return (prep(data.train.data).to(device), data.train.targets.to(device),
            prep(data.val.data).to(device), data.val.targets.to(device))

def show_gray(imgs, num_rows, num_cols, titles=None, scale=1.0):
    """Show images (n, 1, h, w) in [-1, 1], or binary (n, h, w), in gray."""
    imgs = imgs.detach().cpu()
    imgs = ((imgs[:, 0] + 1) / 2).clamp(0, 1) if imgs.is_floating_point() \
        else imgs.float()
    return d2l.show_images(imgs[..., None].expand(-1, -1, -1, 3), num_rows,
                           num_cols, titles, scale)
```

`train_regression` trains the noise predictor, with fresh noise for every minibatch.

```
X, _, X_test, _ = d2l.load_fashion_mnist_pm1()
sigma = 0.5
torch.manual_seed(0)
net = nn.Sequential(nn.Conv2d(1, 32, 3, padding=1), nn.SiLU(),
                    nn.Conv2d(32, 32, 3, padding=1), nn.SiLU(),
                    nn.Conv2d(32, 32, 3, padding=1), nn.SiLU(),
                    nn.Conv2d(32, 1, 3, padding=1)).to(X.device)
```

```
def noise_loss(net, x, sigma):
    """Squared error of the predicted noise; the score is -net / sigma."""
    eps = torch.randn_like(x)
    return ((net(x + sigma * eps) - eps) ** 2).mean()

batch_fn = lambda: X[torch.randint(0, len(X), (128,), device=X.device)]
net, _, losses = d2l.train_regression(
    net, lambda net, x: noise_loss(net, x, sigma), batch_fn, 600)
print(f'last training loss {losses[-50:].mean():.3f}')
```

```
last training loss 0.124
```

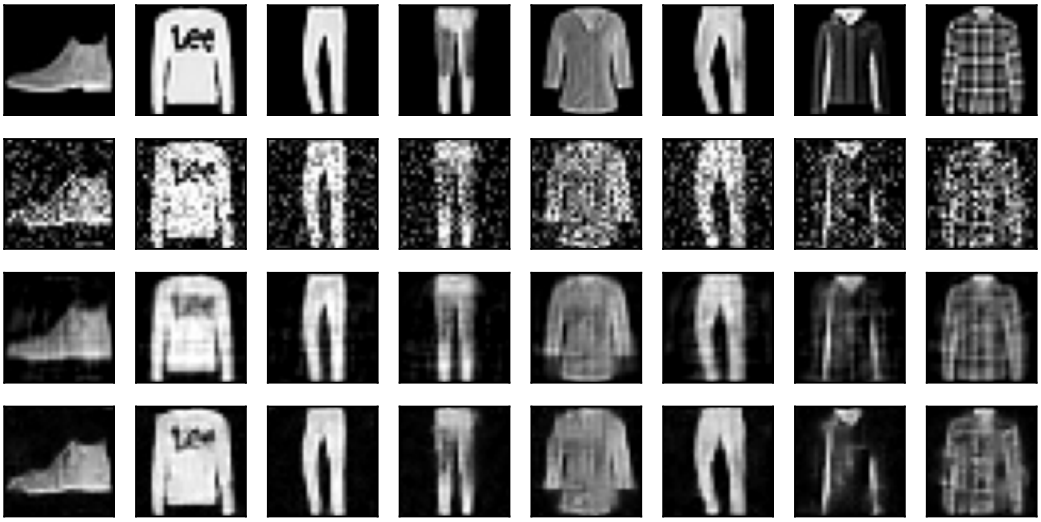
The natural reference is the best *linear* denoiser. If the images were Gaussian with mean μ and covariance Σ, Tweedie's formula would give the shrinkage of the Gaussian case above, along each eigenvector of Σ with its own factor $\lambda/(\lambda+\sigma^2)$:

$$D_{\text{lin}}(\tilde{\mathbf{x}}) = \mu + \Sigma(\Sigma + \sigma^2\mathbf{I})^{-1}(\tilde{\mathbf{x}} - \mu). \tag{17.12}$$

This *Wiener filter* keeps directions in which the images vary more than the noise and suppresses the others; it is the best affine denoiser for any distribution with this mean and covariance (7). The cell estimates μ and Σ from the 60,000 training images and compares the per-pixel squared error of the noisy input, the Wiener filter and the network on 1,000 test images, shown in that order below the clean ones.

```
Xf = X.reshape(len(X), -1)
mu = Xf.mean(0)
Sigma = (Xf - mu).T @ (Xf - mu) / len(Xf)      # (784, 784) pixel covariance
A = torch.linalg.solve(Sigma + sigma ** 2 * torch.eye(784, device=X.device),
                       Sigma)                  # (Sigma + sigma^2 I)^-1 Sigma
torch.manual_seed(1)
with torch.no_grad():
    x = X_test[:1000]
    xt = x + sigma * torch.randn_like(x)
    x_lin = (mu + (xt.reshape(1000, -1) - mu) @ A).reshape(x.shape)
    x_net = xt - sigma * net(xt)               # Tweedie: xt + sigma^2 * score
for name, xh in (('noisy', xt), ('Wiener', x_lin), ('network', x_net)):
    print(f'{name}: per-pixel squared error {((xh - x) ** 2).mean():.4f}')
d2l.show_gray(torch.cat([x[:8], xt[:8], x_lin[:8], x_net[:8]]), 4, 8);
```

```
noisy: per-pixel squared error 0.2493
Wiener: per-pixel squared error 0.0447
network: per-pixel squared error 0.0312
```



The noisy input has error $\sigma^2 = 0.25$ per pixel, the Wiener filter about 0.045, and the network about 0.03, a third less than the best linear map after only 600 steps. The network wins because the images are not Gaussian. A linear filter can only keep or suppress fixed directions of pixel space, the same for every image, and its outputs show blurred edges and a faint grain on the background. The network returns a clean background and sharper edges, and keeps lettering on a sweater that the filter smears: natural images lie near a low-dimensional set inside pixel space (Bengio, Courville, et al. 2013), and a convolutional network can use local shape to decide what is signal.

Both outputs are smooth. A posterior mean averages over every clean image consistent with the noisy one, so texture that the noise destroyed comes back as its average, the blur of Section 16.7. A denoiser returns an average, and a generator must return a draw.

17.1.5 Summary

A network trained by squared error to remove Gaussian noise learns the posterior mean of the clean point; Tweedie's formula makes it a score of the noisy data, and Vincent's theorem makes the regression a way to fit that score. A score needs no normalizer, whereas a density needs an integral that importance sampling cannot estimate in high dimensions. On images, a small convolutional denoiser beats the best linear one.

One noise level cannot both describe the data and sample them. At a small level the regression targets are mostly noise, the score is blind to the mode weights, and Langevin chains never cross between modes. At a large level the chains mix and find the right weights, but they sample a blurred distribution. Section 17.2 uses every noise level at once.

17.1.6 Exercises

1. **Learning without normalization.** Consider the one-dimensional model $\tilde{p}(x) = \exp(-E(x))$ with $E(x) = (x-\mu)^2/(2\tau^2)$, fit to data $x_1, \dots, x_n$ that take at least two distinct values.
    1. Maximize $\sum_i \log \tilde{p}(x_i)$ over μ and τ without normalizing. What does $\tilde{p}$ look like away from the data as the objective approaches its supremum?

   2. Repeat with the normalized model $\tilde{p}/Z$. Which term of the objective prevents the failure of the first part, and what is the maximizer?
   3. A network energy usually ends in a linear layer with an output bias c. Show that the normalized density Equation 17.7 does not depend on c. What does gradient ascent on $-\sum_i E_\theta(\mathbf{x}_i)$ alone do to c?

2. **The log-normal weight.** In the importance-sampling experiment of this section, the target is $\mathcal{N}(\mathbf{m}, \mathbf{I})$ up to normalization and the proposal is $\mathcal{N}(\mathbf{0}, \mathbf{I})$.
   1. Derive $w/Z = \exp(\mathbf{m}^\top \mathbf{x} - \|\mathbf{m}\|^2/2)$ and show with the moment generating function of a Gaussian that $\mathbb{E}_q[w/Z] = 1$ and $\mathbb{E}_q[(w/Z)^2] = \exp(\|\mathbf{m}\|^2)$. Conclude that the relative standard error of the n-sample estimate is $\sqrt{(e^{\|\mathbf{m}\|^2} - 1)/n}$. How many samples give a relative error of 0.1 when $\|\mathbf{m}\|^2 = 20$?
   2. The effective sample size $(\sum_i w_i)^2 / \sum_i w_i^2$ equals n times the squared sample mean of the weights divided by their sample second moment. Use the first part to predict it for large n when $\mathbf{m}$ is the all-ones vector, and compare the prediction with the printed values. Why do the printed values exceed the prediction from $d = 4$ on? What does this imply for the effective sample size as a diagnostic (Section 36.8.2)?

3. **Data on a line.** Let the data lie on the first coordinate axis of $\mathbb{R}^2$, $\mathbf{x}_0 = (u, 0)$ with $u \sim q$ for a smooth positive density q on $\mathbb{R}$, so that p has no density on $\mathbb{R}^2$.
   1. Show that $p_\sigma(\tilde{\mathbf{x}}) = q_\sigma(\tilde{x}_1)\, \mathcal{N}(\tilde{x}_2;\, 0, \sigma^2)$, where q_σ is q blurred by $\mathcal{N}(0, \sigma^2)$, and write the score $\nabla \log p_\sigma$.
   2. Show that the second component of the score has mean square $1/\sigma^2$ under p_σ and slope $-1/\sigma^2$ in $\tilde{x}_2$. What happens to both as $\sigma \to 0$, and what does this mean for a network that must represent the score at a small noise level?
   3. Apply Tweedie's formula Equation 17.5. What does the optimal denoiser do to each coordinate of $\tilde{\mathbf{x}}$, and which geometric operation does it perform?
   4. Let the data lie on a k-dimensional linear subspace of $\mathbb{R}^d$ instead. Compute the mean square of the component of the score orthogonal to the subspace.

4. **Two-point data.** Let $p = \frac{1}{2}\delta_{-1} + \frac{1}{2}\delta_{+1}$ in one dimension, noised at level σ.
   1. Compute the posterior mean $\mathbb{E}[x_0 \mid \tilde{x}]$ by Bayes' rule, and show that it equals $\tilde{x} + \sigma^2 \frac{d}{d\tilde{x}} \log p_\sigma(\tilde{x})$, as Tweedie's formula Equation 17.5 requires.
   2. What does the optimal denoiser return for $\tilde{x} = 0$? Why is this the best answer under squared error even though it is not a possible clean value?
   3. Write the minimum of Equation 17.3 as $\frac{1}{2\sigma^4}\, \mathbb{E}\big[\operatorname{tr} \operatorname{Cov}(\mathbf{x}_0 \mid \tilde{\mathbf{x}})\big]$, and show that it is positive for every data distribution that is not concentrated on a single point.
   4. For the two-point data, write the minimum as a one-dimensional integral over $\tilde{x}$ and evaluate it by quadrature for $\sigma \in \{0.25, 0.5, 1, 2\}$. Multiply each value by σ^2 to express it in the units of the noise-prediction loss. Explain why this rescaled floor vanishes as $\sigma \to 0$, and why it also vanishes as $\sigma \to \infty$.

5. **Bias of the smoothed score.** The text derives the bias $d\sigma^4/(2s^2(s^2+\sigma^2)^2)$ of the smoothed score for one Gaussian component.
   1. Divide it by $\frac{1}{2}\, \mathbb{E}\|\nabla \log p\|^2 = d/(2s^2)$ and show that the ratio depends on σ and s only through σ/s. Its square root is the relative root-mean-square error of the smoothed score. How small must σ/s be for this error to stay below 10%?

2. Compare the formula with the printed bias of the mixture at the seven noise levels of the sweep. Where does the single-component formula stop describing the mixture, and why?
3. What does the rule of the first part imply for data concentrated near a low-dimensional set, as in 3? Which value of s applies to the directions orthogonal to the set?

6. **Laplace noise.** ▨ Vincent's theorem holds for any differentiable noise kernel; Tweedie's formula does not. Perturb one-dimensional data with Laplace noise, $\tilde{x} = x_0 + z$, where z has density $k(z) = e^{-|z|/b}/(2b)$.

   1. Compute the conditional score $\frac{d}{d\tilde{x}} \log k(\tilde{x} - x_0)$ for $\tilde{x} \neq x_0$. Express the minimizer of the corresponding denoising objective in terms of the posterior probability $P(x_0 < \tilde{x} \mid \tilde{x})$.
   2. For the two-point data of 4, compute the score of the perturbed density and the posterior mean $\mathbb{E}[x_0 \mid \tilde{x}]$ for $|\tilde{x}| < 1$ and for $\tilde{x} > 1$. Show that no relation $\mathbb{E}[x_0 \mid \tilde{x}] = \tilde{x} + c\, \frac{d}{d\tilde{x}} \log p_k(\tilde{x})$ with a constant c holds for all $\tilde{x}$, where p_k is the perturbed density.
   3. Which property of the Gaussian conditional score Equation 17.2 makes Tweedie's formula possible?

7. code **The best linear denoiser.**

   1. Show that the Wiener filter Equation 17.12 minimizes $\mathbb{E}\|D(\tilde{\mathbf{x}}) - \mathbf{x}_0\|^2$ among all affine maps D for every data distribution with mean μ and covariance Σ, and that its squared error per pixel is $\frac{1}{d} \sum_i \sigma^2 \lambda_i/(\lambda_i + \sigma^2)$, where $\lambda_1, \dots, \lambda_d$ are the eigenvalues of Σ.
   2. Compute this prediction at $\sigma = 0.5$ from `Sigma` and compare it with the printed error of `x_lin`.
   3. Count the eigenvalues of `Sigma` above σ^2 for $\sigma \in \{0.25, 0.5, 1\}$, and plot a few of the corresponding eigenvectors as images. What does the filter keep at each noise level?

8. code **Step size.** Repeat the Langevin run with the exact score of p at the step sizes $h = 0.0005, 0.005, 0.05, 0.5$ and 1.2. Record the fraction of chains that remain finite, and the summary of the finite chains.

   1. The stationary variance of the discrete chain for a Gaussian of variance v is $v/(1 - h/(4v))$ (Section 38.5.2). Apply `summary` to exact samples of a mixture whose components have this variance instead of 0.25, and compare with the spread and the stray fraction of each run. Which runs does the formula explain, and what happens at $h = 1.2$?
   2. The discretization error is smallest at $h = 0.0005$. Why do the statistics of this run nevertheless differ most from the target?
   3. Among the runs whose chains all remain finite, which one moves the mode weights away from those of the start, and why?

9. code **Escaping a mode.** ▨ Model the valley between the two lower modes of the running example in one dimension by the equal mixture of $\mathcal{N}(-2.5, 0.25)$ and $\mathcal{N}(2.5, 0.25)$, with potential $U = -\log p$, wells at $a = \pm 2.5$, and the barrier at $b = 0$. When the barrier height $\Delta U = U(b) - U(a)$ is large compared with one, Kramers' formula (Kramers 1940) gives the expected time for the diffusion $dX = -\frac{1}{2} U'(X)\, dt + dW$, the continuous-time limit of Equation 17.11, to pass from one

well to the other:

$$\tau \approx \frac{4\pi}{\sqrt{U''(a)\,|U''(b)|}}\, e^{\Delta U}.$$

1. Compute ΔU, $U''(a)$ and $U''(b)$, and convert τ into a number of Langevin steps at $h = 0.05$.
2. Mode 3 of the running example lies closer to mode 1, at a distance of about 4.7. Evaluate $U = -\log p$ of the running example along the segment between the two means, read off the barrier height, and estimate the crossing time in the same way. How many of 2,000 chains should cross from mode 1 to mode 3 in 2,000 steps?
3. Repeat the first part for p_σ at $\sigma = 0.3$ and $\sigma = 1$, treating the result as an order of magnitude where ΔU is small, and use it to explain the weights of the $\sigma = 1$ chains.

10. [extended] **Training an energy by persistent chains.** ⊠⊠ This problem trains an energy-based model Equation 17.7 on the running example by maximum likelihood, the route that this section avoids.

    1. Show that the gradient of the average log-likelihood is $-\mathbb{E}_{\text{data}}[\nabla_\theta E_\theta] + \mathbb{E}_{p_\theta}[\nabla_\theta E_\theta]$.
    2. Let E_θ be an MLP with two hidden layers of 64 units plus $\|\mathbf{x}\|^2/8$, which keeps $\exp(-E_\theta)$ integrable. As a reference, fit it by exact maximum likelihood to `data`, computing Z_θ by quadrature on a 100×100 grid over $[-6, 6]^2$, and report the test negative log-likelihood and the mode weights of the fitted density on the grid (each grid point counted at its nearest mean).
    3. Now estimate the second expectation of the first part with 1,024 persistent chains: each chain keeps its state between updates and takes 20 random-walk Metropolis steps per update, proposing $\mathbf{x}' = \mathbf{x} + 0.25\,\xi$ and accepting with probability $\min\{1, \exp(E_\theta(\mathbf{x}) - E_\theta(\mathbf{x}'))\}$. Train for 3,000 updates and compare the negative log-likelihood and the grid weights with the reference. Why does the acceptance test need no normalizer?
    4. *Contrastive divergence* restarts the chains at the current minibatch in every update and runs one step (Hinton 2002). Train this way and report the same numbers. Which weights does the model learn, and why can chains that never leave the mode they start in carry no information about the weights?

11. code **A denoiser at the wrong noise level.** The network of this section saw only the noise level $\sigma = 0.5$ in training.

    1. Noise 1,000 test images at $\sigma' \in \{0.1, 0.25, 0.5, 0.75, 1\}$ and denoise them with $\tilde{\mathbf{x}} - \sigma'\hat{\epsilon}$. Plot the per-pixel squared error against σ' for the network, for the Wiener filter built for $\sigma = 0.5$, for the Wiener filter recomputed at each σ', and for the noisy input. Where does the network beat the matched Wiener filter, and on which side of 0.5 does it fail? Explain with Tweedie's formula why the best denoiser depends on the noise level, and name the input that would let one network serve every level.
    2. Retrain the network at $\sigma = 10$, where the input is nearly pure noise. Compare its error with that of the Wiener filter at $\sigma = 10$ and with $\text{tr}(\Sigma)/784$, the error of always returning the mean image μ. Show that an error δ in $\hat{\epsilon}$ becomes an error $\sigma\delta$ in the denoised image, and use this to explain what you measure. Which quantity would you rather have the network predict at this noise level?

Adapted from Berkeley CS180/280A, Project 5B (Fall 2025), Sections 1.2.2–1.2.3.

17.2 Gaussian Paths: Diffusion and Flow Matching

The previous section ended with a dilemma. At a small noise level a denoiser is accurate near the data, but Langevin chains driven by its score cannot cross the empty valleys between the modes and keep the mode weights they started with; at a large noise level the chains find the right weights but sample a blurred copy of the data. Suppose that one network denoised at every noise level, and that we walked from pure noise down to the data through all of them. How is such a walk defined, which regression trains the network, and how do we take the steps?

Underneath the three routes by which the literature reached the answer (Chapter 17) is one object: a Gaussian path from the data at $t = 0$ to pure noise at $t = 1$, the posterior mean of the clean point along it, learned by one regression, and a sampler that integrates the differential equation this posterior mean defines.

Current image, video and audio generators are trained this way. We define the path, derive the regression and its equivalent targets, sample with and without noise, and check the learned models against the exact ones.

```
torch.set_float32_matmul_precision('high')  # tf32 matmuls on tensor cores
```

17.2.1 One Path from Data to Noise

A walk through all noise levels needs the levels indexed by a time, which this part defines and tests on the mixture.

17.2.1.1 Signal, Noise and the Log-SNR Clock

We let the data shrink as the noise grows. A *Gaussian path* is a pair of functions α_t and σ_t on $t \in [0, 1]$ that defines the noisy point

$$\mathbf{x}_t = \alpha_t \, \mathbf{x}_0 + \sigma_t \, \epsilon, \qquad \epsilon \sim \mathcal{N}(\mathbf{0}, \mathbf{I}), \tag{17.13}$$

with $(\alpha_0, \sigma_0) = (1, 0)$ and $(\alpha_1, \sigma_1) = (0, 1)$, or nearly so. The *variance-preserving* (VP) path keeps $\alpha_t^2 + \sigma_t^2 = 1$, so data of unit variance keep unit variance; it is the Ornstein–Uhlenbeck process of Section 38.3.3. We write $\bar{\alpha}_t := \alpha_t^2$ and use the continuous form of the schedule of Ho et al. (2020), $\log \bar{\alpha}_t = -(\beta_{\min} t + (\beta_{\max} - \beta_{\min})\, t^2/2)$ with $\beta_{\min} = 0.1$ and $\beta_{\max} = 20$. The *linear* path of flow matching and rectified flow (Lipman et al. 2023; Liu et al. 2023; Albergo et al. 2023) draws a straight line from the data to the noise,

$$\alpha_t = 1 - t, \qquad \sigma_t = t. \tag{17.14}$$

What a denoiser can infer from $\mathbf{x}_t$ depends only on the ratio of signal to noise, measured on a log scale as the *log-SNR* $\lambda_t = \log(\alpha_t^2/\sigma_t^2)$. It falls monotonically from $+\infty$ at the data to $-\infty$ at the noise, a clock that all paths share: at times of equal λ, two paths hold the same noisy point up to a scale factor (7). The VP path reaches $\bar{\alpha}_1 = e^{-10.05} \approx 4 \cdot 10^{-5}$ at $t = 1$, close enough to pure noise (Figure 17.4).

One clock

Data sit at $t = 0$ and noise at $t = 1$ throughout the book (Section 38.6), and every sampler runs from $t = 1$ down to 0. Flow-matching papers reverse time (Figure 38.25): substitute $1 - t$ for their t, read their noise $\mathbf{x}_0$ as our ϵ and their data $\mathbf{x}_1$ as our $\mathbf{x}_0$, and flip the sign of every velocity.

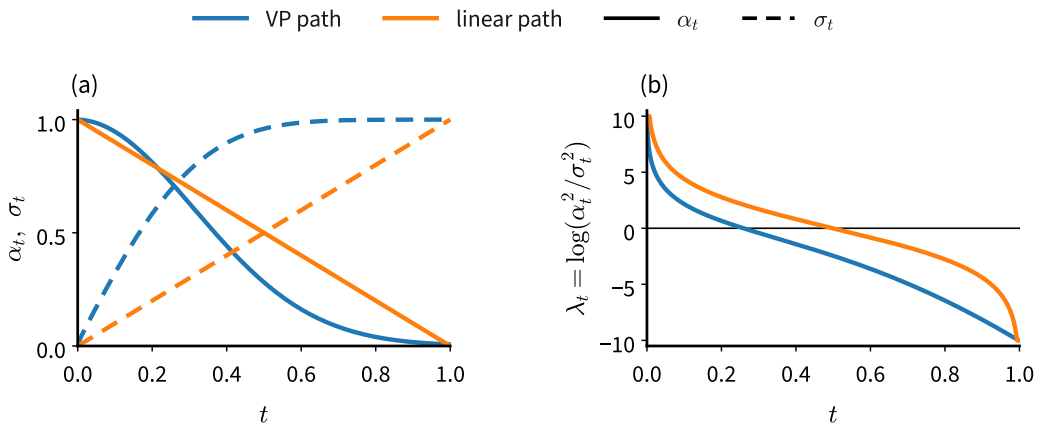


Figure 17.4: Two Gaussian paths on one clock. (a) The signal coefficient α_t (solid) and the noise coefficient σ_t (dashed) of the variance-preserving path, with β from 0.1 to 20, and of the linear path. (b) Their log-SNR λ_t, clipped to $[-10, 10]$. The variance-preserving path reaches the noisy regime much earlier.

`GaussianPath` holds the two functions α_t and σ_t, the log-SNR and a draw of $\mathbf{x}_t$; `vp`, `linear` and `cosine` build the paths of this section.

```
class GaussianPath:
    """x_t = alpha(t) x0 + sigma(t) eps, data at t = 0 and noise at t = 1."""
    def __init__(self, alpha, sigma):
        self.alpha, self.sigma = alpha, sigma

    @staticmethod
    def vp(beta_min=0.1, beta_max=20.0):
        """Variance preserving, Ho et al.'s linear beta in continuous time."""
        log_abar = lambda t: -(beta_min * t + (beta_max - beta_min) * t**2 / 2)
        return GaussianPath(lambda t: torch.exp(log_abar(t) / 2),
                            lambda t: torch.sqrt(-torch.expm1(log_abar(t))))

    @staticmethod
    def linear():
        """The straight line of flow matching and rectified flow."""
        return GaussianPath(lambda t: 1 - t, lambda t: t)

    @staticmethod
    def cosine(s=0.008):
```

```
        """Variance preserving, Nichol and Dhariwal's cosine schedule."""
        c = lambda t: torch.cos(math.pi / 2 * (t + s) / (1 + s))
        g = lambda t: c(t) / c(torch.zeros_like(t))   # exactly 1 at t = 0
        # and exactly 0 at t = 1 (cos(pi/2) is not), keeping the slope
        f = lambda t: torch.where(t < 1, g(t), g(t) - g(t).detach())
        return GaussianPath(f, lambda t: torch.sqrt(1 - f(t) ** 2))

    def log_snr(self, t):
        """lambda_t = log(alpha_t^2 / sigma_t^2), falling from +inf at 0."""
        return 2 * (torch.log(self.alpha(t)) - torch.log(self.sigma(t)))

    def sample(self, x0, t, eps):
        """x_t for per-example times t of shape (n,)."""
        t = t.reshape(-1, *[1] * (x0.dim() - 1))
        return self.alpha(t) * x0 + self.sigma(t) * eps
```

We print both coefficients and the log-SNR at three times.

```
vp, linear = GaussianPath.vp(), GaussianPath.linear()
t = torch.tensor([0.1, 0.5, 0.9])
r2 = lambda v: [round(u, 2) for u in v.tolist()]
for name, path in (('vp', vp), ('linear', linear)):
    print(f'{name:>6}: alpha {r2(path.alpha(t))} sigma {r2(path.sigma(t))} '
          f'log-SNR {r2(path.log_snr(t))}')
```

```
    vp: alpha [0.95, 0.28, 0.02] sigma [0.32, 0.96, 1.0] log-SNR [2.16, -2.46, -8.15]
linear: alpha [0.9, 0.5, 0.1] sigma [0.1, 0.5, 0.9] log-SNR [4.39, 0.0, -4.39]
```

Halfway along in time the VP path is already mostly noise, with $\bar{\alpha} = 0.079$ and $\lambda \approx -2.5$ against $\lambda = 0$ on the linear path, and at $t = 0.9$ it keeps less than two percent of the signal against ten percent.

17.2.1.2 The Chain View

The forward process of Ho et al. (2020) is a Markov chain of T small steps, each shrinking the current point and adding a little noise,

$$\mathbf{x}_k = \sqrt{1-\beta_k}\,\mathbf{x}_{k-1} + \sqrt{\beta_k}\,\epsilon_k, \qquad k = 1, \dots, T, \tag{17.15}$$

with independent standard Gaussians ϵ_k and a variance schedule $\beta_k \in (0, 1)$. Each step preserves unit variance, and because the noises are independent Gaussians, consecutive steps compose into one: the marginal of every step is a single draw,

$$\mathbf{x}_k = \sqrt{\bar{\alpha}_k}\,\mathbf{x}_0 + \sqrt{1-\bar{\alpha}_k}\,\epsilon, \qquad \bar{\alpha}_k = \prod_{j=1}^{k}(1-\beta_j), \tag{17.16}$$

which Section 38.5.3.2 proves by induction. With $\beta_k = 1 - \bar{\alpha}_{t_k}/\bar{\alpha}_{t_{k-1}}$ the DDPM step is the exact transition of the VP path from t_{k-1} to t_k, so the chain is the VP path read on the grid $t_k = k/T$. Training never simulates the chain, because any $\mathbf{x}_k$ is one Gaussian draw away from $\mathbf{x}_0$. Figure 17.5 shows the forward chain and the reverse chain that generation runs.

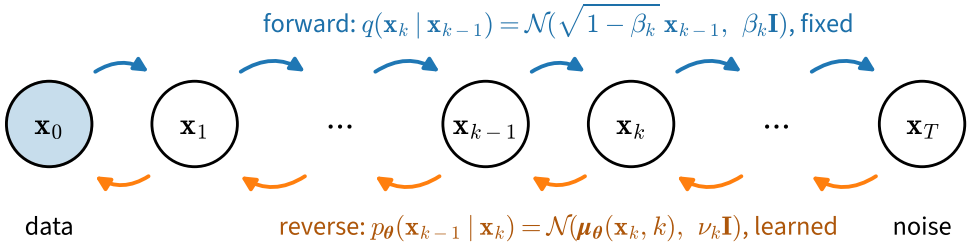


Figure 17.5: The graphical model of a diffusion model. The forward process (top) is a fixed Markov chain of Gaussian transitions from the data $\mathbf{x}_0$ to noise $\mathbf{x}_T$; the reverse process (bottom) is a learned chain of Gaussian transitions whose means a network predicts.

`grid` evaluates the path at even times from noise down to data, the grid every sampler steps along; a start time below 1 begins partway along the path.

```
@d2l.add_to_class(GaussianPath)
def grid(self, steps, t_start=1.0, t_end=0.0):
    """Lists of alpha and sigma at steps + 1 even times, t_start to t_end."""
    t = torch.linspace(t_start, t_end, steps + 1, dtype=torch.float64)
    return self.alpha(t).tolist(), self.sigma(t).tolist()
```

Which schedule does our VP path define on a grid of $T = 1000$ steps? The grid lists the times from the noise end, so its first entries belong to the last step of the chain.

```
a, _ = vp.grid(1000)
print(f'beta_1 = {1 - (a[-2] / a[-1]) ** 2:.1e}, '
      f'beta_T = {1 - (a[0] / a[1]) ** 2:.4f}, alpha_bar_T = {a[0] ** 2:.1e}')
```

```
beta_1 = 1.1e-04, beta_T = 0.0198, alpha_bar_T = 4.3e-05
```

The steps grow from about 10^{-4} at the data to 0.02 at the noise, the linear schedule of Ho et al. (2020), and after a thousand steps only a factor $\sqrt{\bar{\alpha}_T} \approx 0.007$ of the data survives.

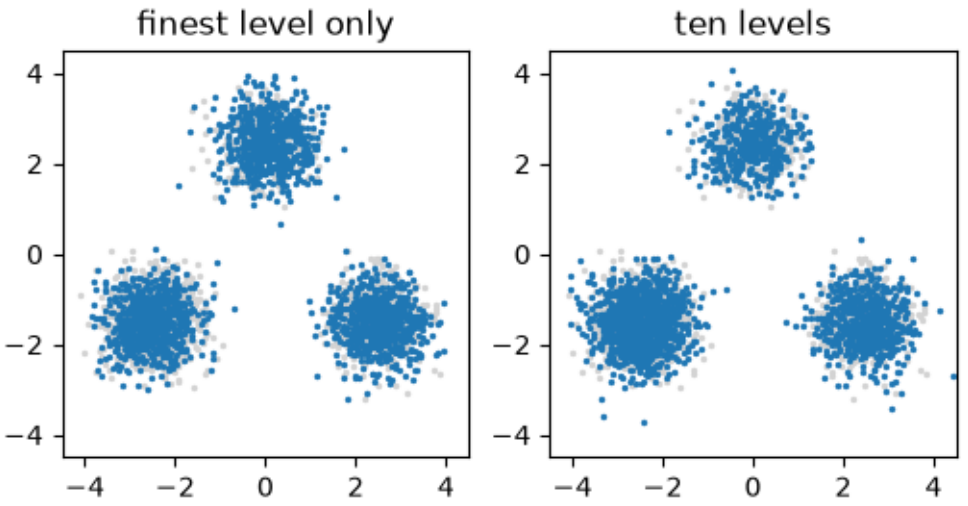
17.2.1.3 Annealing through Noise Levels

Does a walk through many noise levels do what one level could not? Because the scores of the smoothed mixture are known exactly, we can test this before training any network. We run the Langevin sampler Equation 17.11 down a *ladder* of noise levels: ten levels $\tilde{\sigma}_i$ spaced evenly in log scale from 3 down to 0.05, each with 100 steps of size $h_i = 0.1\,\tilde{\sigma}_i^2$ on the exact score of $p_{\tilde{\sigma}_i}$, starting where the previous level stopped. The control spends the same $1{,}000$ steps at the finest level, with the step size $h = 0.05$ of the previous section. Both start from the same $2{,}000$ points drawn from $\mathcal{N}(\mathbf{0}, 9\mathbf{I})$.

```
torch.manual_seed(0)
device = d2l.try_gpu()
mix = d2l.GaussianMixture.three_modes(device)
x_init = 3 * torch.randn(2000, 2, device=device)
```

```
x = x_init
for s in torch.logspace(math.log10(3), math.log10(0.05), 10).tolist():
    x = d2l.langevin(lambda z: mix.score(z, 1.0, s), x, 0.1 * s ** 2, 100)
control = d2l.langevin(lambda z: mix.score(z, 1.0, 0.05), x_init, 0.05, 1000)
print('finest level only:', mix.summary(control))
print('ten levels:       ', mix.summary(x))
d2l.scatter_panels([control, x], ['finest level only', 'ten levels'],
                   data=mix.sample(2000));
```

```
finest level only: {'weights': [0.339, 0.331, 0.33], 'stray': 0.0145, 'spread': 0.493}
ten levels:        {'weights': [0.496, 0.297, 0.208], 'stray': 0.015, 'spread': 0.516}
```



The ladder lands within about 0.03 of the weights $0.5/0.3/0.2$, with one to two percent of the points straying between the modes. The control has the right spread around each mode but keeps the weights of its starting points, about a third each. The proportions are decided at the coarse levels, where the smoothed density has no valleys; the fine levels only sharpen each mode. A diffusion model divides the work the same way along its path.

This *annealed Langevin dynamics* sampled the first generator of this kind (Song and Ermon 2019), one network that takes the noise level as input, trained by denoising score matching at every level of the ladder with weights $\tilde{\sigma}_i^2$ (Section 38.5.3, 3). Its levels, step sizes and step counts were designed by hand (8). The path replaces the ladder by a continuum of levels, and a sampler that the path determines replaces the hand-tuned Langevin steps.

17.2.2 One Regression

One network is to serve every point of the path. What should it learn at a noisy point, which regression teaches it, and how do the targets of the literature relate?

17.2.2.1 Conditional Targets and Marginal Fields

The object to learn is again the posterior mean of the clean point, $\hat{\mathbf{x}}_0 = \mathbb{E}[\mathbf{x}_0 \mid \mathbf{x}_t]$, which gave the score by Tweedie's formula Equation 17.5 and also tells a noisy point which way to

move. It depends on the whole data distribution and is unknown, but its conditional version is trivial: given $\mathbf{x}_0$, the best guess of $\mathbf{x}_0$ is $\mathbf{x}_0$ itself.

The same holds for motion along the path. A single noisy point moves along $\mathbf{x}_t = \alpha_t \mathbf{x}_0 + \sigma_t \epsilon$ with the *conditional velocity* $\dot{\mathbf{x}}_t = \dot{\alpha}_t \mathbf{x}_0 + \dot{\sigma}_t \epsilon$, which on the linear path is $\epsilon - \mathbf{x}_0$. The whole cloud of noisy points moves at $\mathbf{x}$ with the average velocity of the points that pass there,

$$\mathbf{u}_t(\mathbf{x}) = \mathbb{E}\big[\dot{\alpha}_t\, \mathbf{x}_0 + \dot{\sigma}_t\, \epsilon \,\big|\, \mathbf{x}_t = \mathbf{x}\big], \tag{17.17}$$

the *marginal velocity*. Where only one clean point can have produced $\mathbf{x}$, it is that point's velocity; where several can, it is their average, weighted by how likely each is.

A field defined as a conditional expectation is learned by regressing on the quantity inside it, because least squares recovers the conditional mean; this is how denoising score matching recovered the score. For the velocity it gives the *conditional flow matching* loss, which differs from the regression onto the unknown marginal field only by a constant C that does not depend on the network:

$$\mathbb{E}_{t,\,\mathbf{x}_0,\,\epsilon} \left\| \mathbf{u}_\theta(\mathbf{x}_t, t) - (\dot{\alpha}_t \mathbf{x}_0 + \dot{\sigma}_t \epsilon) \right\|^2 = \mathbb{E}_{t,\,\mathbf{x}_t} \left\| \mathbf{u}_\theta(\mathbf{x}_t, t) - \mathbf{u}_t(\mathbf{x}_t) \right\|^2 + C. \tag{17.18}$$

The two sides have the same minimizer, so a network trained on the per-example velocity learns the marginal one (Section 38.6.1). Figure 17.6 shows both views.

`target` gives the per-example regression target of a kind on any path. For `'velocity'` it is the conditional velocity, with $\dot{\alpha}_t$ and $\dot{\sigma}_t$ from forward-mode differentiation (`rates`); `x0_eps` turns a prediction of any kind back into the pair $(\hat{\mathbf{x}}_0, \hat{\epsilon})$.

```
@d2l.add_to_class(GaussianPath)
def rates(self, t):
    """The time derivatives (alpha'(t), sigma'(t)), by forward-mode AD."""
    ones = torch.ones_like(t)
    return (torch.func.jvp(self.alpha, (t,), (ones,))[1],
            torch.func.jvp(self.sigma, (t,), (ones,))[1])

@d2l.add_to_class(GaussianPath)
def target(self, kind, x0, eps, t):
    """The regression target of a kind: eps, x0, v, score or velocity."""
    t = t.reshape(-1, *[1] * (x0.dim() - 1))
    a, s = self.alpha(t), self.sigma(t)
    if kind == 'eps': return eps
    if kind == 'x0': return x0
    if kind == 'v': return a * eps - s * x0
    if kind == 'score': return -eps / s
    if kind != 'velocity': raise ValueError(f'unknown kind {kind!r}')
    da, ds = self.rates(t)                      # velocity = d x_t / dt
    return da * x0 + ds * eps

@d2l.add_to_class(GaussianPath)
def x0_eps(self, kind, out, x, t):
    """The (x0, eps) that a prediction `out` of a kind implies at (x, t)."""
    t = t.reshape(-1, *[1] * (x.dim() - 1))
    a, s = self.alpha(t), self.sigma(t)
```

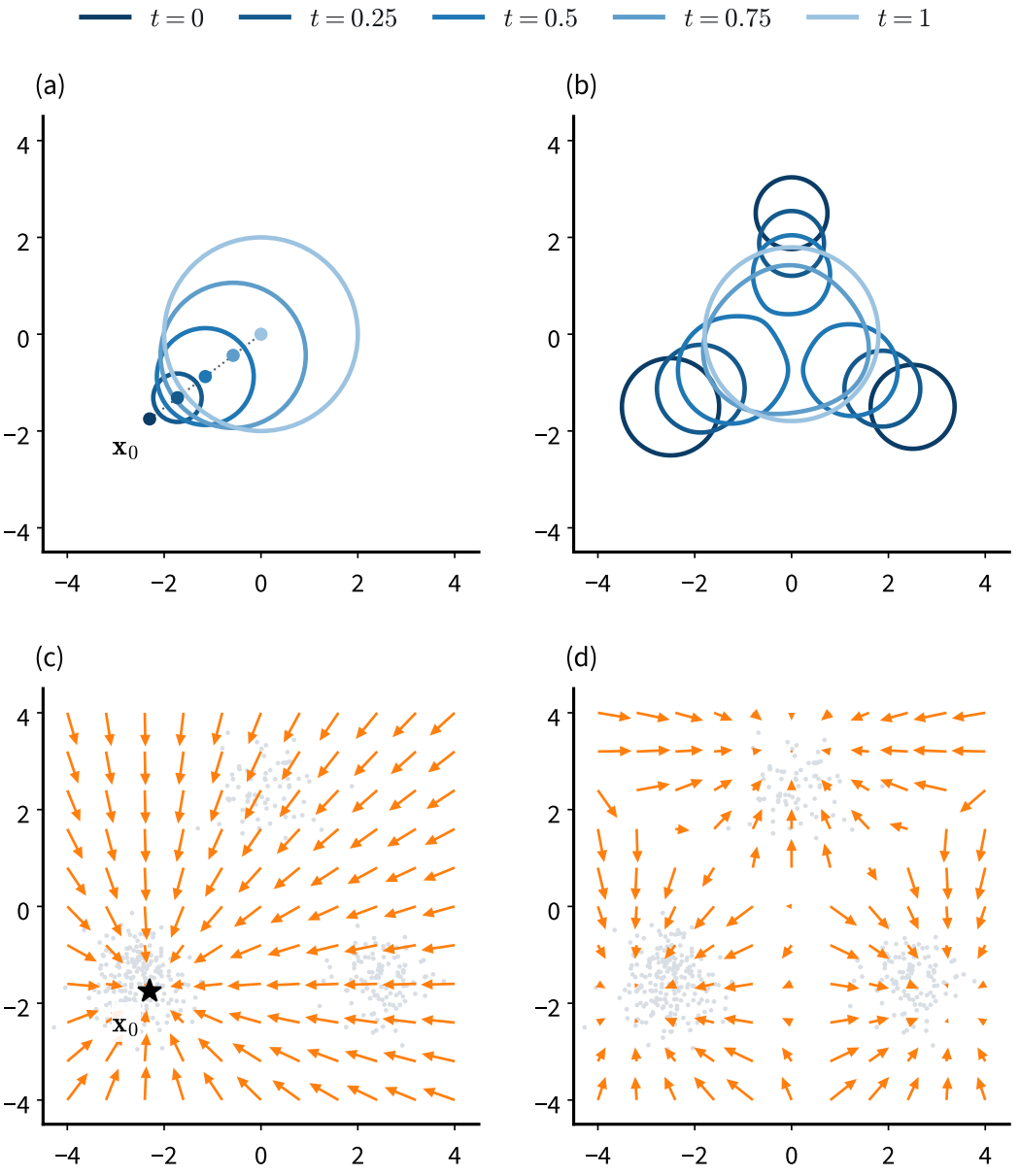


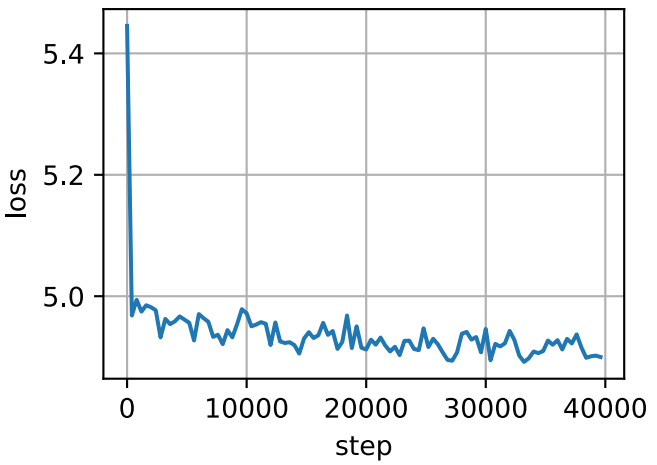
Figure 17.6: The conditional and the marginal views of the linear path on the mixture, computed exactly. (a) The conditional path of one clean point $\mathbf{x}_0$ from the heaviest mode: circles of radius $2\sigma_t$ around $\alpha_t \mathbf{x}_0$ at five times from 0 to 1. (b) The marginal path: contours of p_t at the same times. (c) The conditional field $-(\mathbf{x} - \mathbf{x}_0)/t$ at $t = 0.5$, pointing toward that one point. (d) The marginal field $-\mathbf{u}_{0.5}(\mathbf{x})$, the posterior average of the conditional fields.

```
    if kind == 'eps': return (x - s * out) / a, out
    if kind == 'x0': return out, (x - a * out) / s
    if kind == 'v': return ((a * x - s * out) / (a**2 + s**2),
                            (s * x + a * out) / (a**2 + s**2))
    if kind == 'score': return (x + s ** 2 * out) / a, -s * out
    if kind != 'velocity': raise ValueError(f'unknown kind {kind!r}')
    da, ds = self.rates(t)
    det = a * ds - s * da
    return (ds * x - s * out) / det, (a * out - da * x) / det
```

The method fits in one cell. Each step draws 512 fresh points from the mixture and a time and a noise for each, and regresses `TimeMLP` onto the conditional velocity of the linear path, for $40{,}000$ Adam steps with a cosine-decayed learning rate; `train_regression` also returns a moving average of the weights with a half-life of 700 steps.

```
def regression_loss(path, kind):
    """Squared error of net(x_t, t) against the target of a kind, t ~ U(0, 1)."""
    def loss_fn(net, x0):
        t = torch.rand(len(x0), device=x0.device)
        eps = torch.randn_like(x0)
        error = net(path.sample(x0, t, eps), t) - path.target(kind, x0, eps, t)
        return (error ** 2).flatten(1).sum(1).mean()  # ||error||^2 per example
    return loss_fn

torch.manual_seed(1)
num_steps = 40_000
net_v, ema_v, losses_v = d2l.train_regression(
    d2l.TimeMLP(256, 3).to(device), regression_loss(linear, 'velocity'),
    lambda: mix.sample(512), num_steps, cosine=True, ema_half_life=700)
d2l.plot(torch.arange(100) * num_steps // 100,
         losses_v.reshape(100, -1).mean(1).cpu(), 'step', 'loss')
```



Each point of the curve averages 400 steps. The loss falls within a few thousand steps and levels off near 5, far above zero; we will see that no network can do much better.

17.2.2.2 Four Targets, One Oracle

The network above predicts a velocity; DDPM's predicts the noise, and others predict the clean point. At a fixed $\mathbf{x}_t$ these are not different quantities. Taking the posterior mean of Equation 17.13 gives $\mathbf{x}_t = \alpha_t \hat{\mathbf{x}}_0 + \sigma_t \hat{\epsilon}$, so $\hat{\epsilon} = \mathbb{E}[\epsilon \mid \mathbf{x}_t]$ and $\hat{\mathbf{x}}_0$ determine each other, and Tweedie's formula for the kernel $\mathcal{N}(\alpha_t \mathbf{x}_0, \sigma_t^2 \mathbf{I})$ gives the score, as in Equation 17.6:

$$\nabla \log p_t(\mathbf{x}_t) = \frac{\alpha_t \hat{\mathbf{x}}_0 - \mathbf{x}_t}{\sigma_t^2} = -\frac{\hat{\epsilon}}{\sigma_t}. \tag{17.19}$$

The marginal velocity Equation 17.17 is $\dot{\alpha}_t \hat{\mathbf{x}}_0 + \dot{\sigma}_t \hat{\epsilon}$, and eliminating $\hat{\mathbf{x}}_0$ between it and the score gives the score–velocity identity

$$\mathbf{u}_t(\mathbf{x}) = \frac{\dot{\alpha}_t}{\alpha_t}\,\mathbf{x} - \left(\sigma_t \dot{\sigma}_t - \sigma_t^2 \frac{\dot{\alpha}_t}{\alpha_t}\right) \nabla \log p_t(\mathbf{x}), \tag{17.20}$$

with multiples fixed by the path. On the linear path it reads

$$\mathbf{u}_t(\mathbf{x}) = \hat{\epsilon} - \hat{\mathbf{x}}_0 = -\frac{\mathbf{x} + t\,\nabla \log p_t(\mathbf{x})}{1 - t}. \tag{17.21}$$

A fifth target, $\mathbf{v} = \alpha_t \epsilon - \sigma_t \mathbf{x}_0$, was introduced by Salimans and Ho (2022) as $\mathbf{v}$-*prediction*; on the cosine path $\alpha_t = \cos \frac{\pi t}{2}$, $\sigma_t = \sin \frac{\pi t}{2}$ it equals $2/\pi$ times the velocity. All of these are affine functions of one posterior mean, the *oracle* that a perfect network would learn, so a network trained on any of them defines all of them (Table 38.2 collects the conversions). The named models of the literature are choices of path and target (Table 17.1).

DDPM's training loss, which Ho et al. (2020) called the *simple loss*, is

$$L_{\text{simple}}(\theta) = \mathbb{E}_{t,\,\mathbf{x}_0,\,\epsilon} \left\| \epsilon - \epsilon_\theta(\alpha_t \mathbf{x}_0 + \sigma_t \epsilon,\, t) \right\|^2. \tag{17.22}$$

Since the conditional score of the kernel is $-\epsilon/\sigma_t$, the simple loss is denoising score matching Equation 17.3 at every level, weighted by σ_t^2 (Section 38.5.3.2). Ho et al. derived it from a variational bound on the log-likelihood; with one particular weighting over t the regression is such a bound (Section 17.5).

`target` and `x0_eps` above implement these conversions. `make_pred` wraps a trained network as the pair $(\hat{\mathbf{x}}_0, \hat{\epsilon})$ at any (α, σ), by querying it at the time of its own path with the same log-SNR, which `time_at` finds by bisection. An ϵ-predictor has no $\hat{\mathbf{x}}_0 = (\mathbf{x} - \sigma\hat{\epsilon})/\alpha$ at pure noise, $\alpha = 0$, so samplers that start there on the linear and cosine paths need another kind; an ϵ-model samples on the VP path, whose α stays positive.

```
@d2l.add_to_class(GaussianPath)
def time_at(self, log_snr):
    """The time with a given log-SNR, by bisection (lambda_t falls in t)."""
    lo, hi = torch.zeros_like(log_snr), torch.ones_like(log_snr)
    for _ in range(50):
        mid = (lo + hi) / 2
        later = self.log_snr(mid) > log_snr      # the answer lies after mid
        lo, hi = torch.where(later, mid, lo), torch.where(later, hi, mid)
```

```
    return (lo + hi) / 2

def make_pred(net, path, kind):
    """pred(x, alpha, sigma) -> (x0, eps) from net(x, t) of a kind on path."""
    def pred(x, alpha, sigma):
        log_snr = 2 * math.log(alpha / sigma) if alpha > 0 else -math.inf
        t = path.time_at(torch.tensor(log_snr, dtype=torch.float64))
        x = x * (path.sigma(t).item() / sigma)  # the same point on our path
        t = torch.full((len(x),), t.item(), device=x.device)
        return path.x0_eps(kind, net(x, t), x, t)
    return pred
```

The second model makes DDPM's choice, the noise target on the VP path, and changes nothing else.

```
torch.manual_seed(2)
net_eps, ema_eps, losses_eps = d2l.train_regression(
    d2l.TimeMLP(256, 3).to(device), regression_loss(vp, 'eps'),
    lambda: mix.sample(512), num_steps, cosine=True, ema_half_life=700)
```

The two networks differ in path and target, but by the dictionary they should hold the same posterior mean, which the last part of the section checks.

17.2.2.3 The Loss Floor

The velocity model's loss levelled off near 5 for the reason of Section 17.1: a target contains the noise of its own example, so even the oracle keeps the conditional variance of the target given $\mathbf{x}_t$, now averaged over the path. For the mixture this floor is exact. The function `oracle` takes $\hat{\mathbf{x}}_0$ from `GaussianMixture.posterior_mean` and $\hat{\epsilon}$ from $\mathbf{x}_t = \alpha_t \hat{\mathbf{x}}_0 + \sigma_t \hat{\epsilon}$; every target is the same affine function of them, and any sampler can use `oracle` in place of a network.

```
def oracle(x, alpha, sigma):
    """The exact posterior means (x0_hat, eps_hat) of the mixture at x."""
    x0_hat = mix.posterior_mean(x, alpha, sigma)
    a, s = (torch.as_tensor(v, device=x.device).reshape(-1, 1)
            for v in (alpha, sigma))
    return x0_hat, (x - a * x0_hat) / s

def loss_floor(path, kind, x0, eps, t):
    """E||target - E[target | x_t]||^2 over the draws (x0, eps, t)."""
    x = path.sample(x0, t, eps)
    best = path.target(kind, *oracle(x, path.alpha(t), path.sigma(t)), t)
    return ((path.target(kind, x0, eps, t) - best) ** 2).sum(1).mean().item()

n = 1_000_000
x0, eps = mix.sample(n), torch.randn(n, 2, device=device)
times = [torch.full((n,), t, device=device) for t in (0.1, 0.5, 0.9)]
times.append(torch.rand(n, device=device))             # t ~ U(0, 1)
print('target, path      t=0.1  t=0.5  t=0.9  t~U(0,1)  trained')
for name, path, kind, losses in (('noise, vp', vp, 'eps', losses_eps),
        ('velocity, linear', linear, 'velocity', losses_v)):
```

```
    floors = [loss_floor(path, kind, x0, eps, t) for t in times]
    print(f'{name:16}' + ''.join(f'{f:7.3f}' for f in floors[:3])
          + f'{floors[3]:10.3f}{losses[-1000:].mean().item():9.3f}')
```

```
target, path      t=0.1  t=0.5  t=0.9  t~U(0,1)  trained
noise, vp         1.368  0.477  0.002     0.540    0.542
velocity, linear  2.351  2.796  9.157     4.906    4.899
```

Each training loss sits at its floor. The floors differ by an order of magnitude, about 0.5 for the noise and about 5 for the velocity, because the targets are scaled differently. The noise floor is largest near the data, where noise and the spread of a mode look alike; the velocity floor, $\mathrm{Var}(\mathbf{x}_0 \mid \mathbf{x}_t)/t^2$ summed over coordinates, grows toward the noise end, where the clean point is hardest to guess. A score target would divide the noise floor by σ_t^2, without bound near the data (4). Losses of different targets cannot be compared; predictors trained on different targets can, through the dictionary.

17.2.3 Sampling by Integration

The marginal velocity carries the noise distribution to the data distribution, so we sample by following it, along deterministic trajectories or with fresh noise at every step.

17.2.3.1 The Probability-Flow ODE and Euler's Method

The pair $(p_t, \mathbf{u}_t)$ satisfies the continuity equation (Section 38.6.1): the marginal velocity moves the density p_t of $\mathbf{x}_t$ along the path. A point drawn from $p_1 = \mathcal{N}(\mathbf{0}, \mathbf{I})$ and carried backward in time by the ordinary differential equation

$$\frac{d\mathbf{x}}{dt} = \mathbf{u}_t(\mathbf{x}), \qquad \text{from } t = 1 \text{ down to } t = 0, \tag{17.23}$$

is therefore a draw from p_t at every time, and from the data at $t = 0$. This *probability-flow ODE* is the deterministic twin of the diffusion process (Section 38.4.4.3). We build it up from the smallest case, as Nakkiran et al. (2025) do.

Let the data be one point $\mathbf{x}_0$. On the linear path a point at $\mathbf{x}$ carries the noise $\epsilon = (\mathbf{x} - (1-t)\mathbf{x}_0)/t$ and moves with velocity $\epsilon - \mathbf{x}_0 = (\mathbf{x} - \mathbf{x}_0)/t$. Every trajectory is a straight line into $\mathbf{x}_0$, and a single step of length one from $t = 1$, $\mathbf{x} \leftarrow \mathbf{x} - \mathbf{u}_1(\mathbf{x}) = \mathbf{x}_0$, is exact. With two data points the field is the posterior-weighted average of two such straight fields, and the trajectories bend where the posterior switches from one point to the other (Section 38.6.1.3).

Any distribution is a mixture of point masses, with straight conditional paths and a curved marginal flow. At $t = 1$ the point is pure noise, so the posterior equals the prior and $\mathbf{u}_1(\mathbf{x}) = \mathbf{x} - \mathbb{E}[\mathbf{x}_0]$: one step of length one sends every start to the mean of the data (Section 38.6.1.3). For our mixture that mean is $(-0.5, -0.7)$, between the modes, where no data lie.

Euler's method (Section 38.1.3) follows the curve in K straight steps, $\mathbf{x} \leftarrow \mathbf{x} - \mathbf{u}_t(\mathbf{x})/K$ at $t = 1, 1 - 1/K, \ldots, 1/K$. We run it with the learned field from $2{,}000$ standard Gaussian starts.

```
@torch.no_grad()
def euler(net, x, steps):
```

```
    """Euler's method on dx/dt = net(x, t), from t = 1 down to t = 0."""
    for k in range(steps, 0, -1):
        x = x - net(x, torch.full((len(x),), k / steps, device=x.device)) / steps
    return x

torch.manual_seed(3)
x_T = torch.randn(2000, 2, device=device)
print(mix.summary(euler(ema_v, x_T, 100)))
```

```
{'weights': [0.514, 0.277, 0.21], 'stray': 0.0135, 'spread': 0.471}
```

A hundred steps put the samples on the three modes. The library's sampler takes the same steps in other coordinates: along every Gaussian path, $\bar{\mathbf{x}} = \mathbf{x}/\alpha_t$ and the noise-to-signal ratio $\bar{\sigma} = \sigma_t/\alpha_t$ obey $d\bar{\mathbf{x}}/d\bar{\sigma} = \hat{\epsilon}$ (Karras et al. 2022), a form that serves every path and predictor.

`ode_sample` takes Euler steps of this equation through a list of levels $\bar{\sigma}$, here the path's grid in t from `sigma_bars`; second-order steps and other grids wait for Section 17.4.

```
@d2l.add_to_class(GaussianPath)
def sigma_bars(self, steps):
    """sigma_bar = sigma / alpha on the grid of steps; inf where alpha = 0."""
    a, s = self.grid(steps)
    return [s_k / a_k if a_k > 0 else math.inf for a_k, s_k in zip(a, s)]

def pred_bar(pred, xb, r):
    """pred at x = alpha x_bar for sigma_bar = r; at r = inf, x_bar is noise."""
    if math.isinf(r):
        return pred(xb, 0.0, 1.0)
    a = 1 / math.sqrt(1 + r ** 2)
    return pred(xb * a, a, r * a)

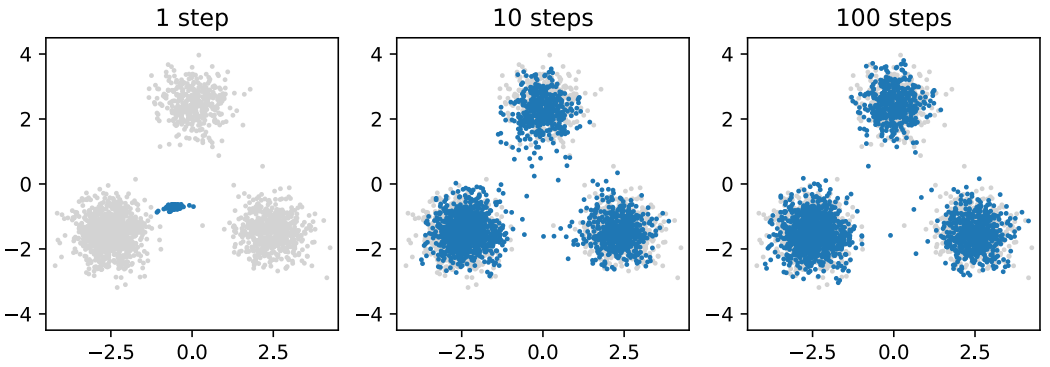
@torch.no_grad()
def ode_sample(pred, x, sigma_bars):
    """Euler (= DDIM) on d x_bar / d sigma_bar = eps, from x to data; x is the
    noisy point at the first level, unit-variance scale (noise if inf)."""
    r0 = sigma_bars[0]
    xb = x if math.isinf(r0) else x * math.sqrt(1 + r0 ** 2)  # x / alpha
    for r_t, r_s in zip(sigma_bars[:-1], sigma_bars[1:]):
        x0, eps = pred_bar(pred, xb, r_t)
        xb = x0 + r_s * eps                    # the Euler (= DDIM) step
    return xb
```

How many steps does the flow need? We integrate the wrapped velocity network from the same starts with one, ten and a hundred steps, and check that a hundred steps of `ode_sample` reproduce the loop above.

```
pred_v = make_pred(ema_v, linear, 'velocity')
samples = [ode_sample(pred_v, x_T, linear.sigma_bars(steps))
           for steps in (1, 10, 100)]
print('one step: mean', [round(v, 3) for v in samples[0].mean(0).tolist()],
```

```
        'std', [round(v, 3) for v in samples[0].std(0).tolist()])
for steps, x in zip((10, 100), samples[1:]):
    print(f'{steps} steps:', mix.summary(x))
print('largest difference to the loop:',
      (samples[2] - euler(ema_v, x_T, 100)).abs().max().item())
d2l.scatter_panels(samples, ['1 step', '10 steps', '100 steps'],
                   data=mix.sample(2000));
```

```
one step: mean [-0.52, -0.715] std [0.044, 0.022]
10 steps: {'weights': [0.522, 0.273, 0.206], 'stray': 0.025, 'spread': 0.448}
100 steps: {'weights': [0.514, 0.277, 0.21], 'stray': 0.0135, 'spread': 0.471}
largest difference to the loop: 0.00012962520122528076
```



One step lands every start on the data mean $(-0.5, -0.7)$, with almost no spread, as the point-mass argument predicts. Ten steps recover the three modes with weights within about 0.03 of the targets, but about three percent of the samples stray between the modes. A hundred steps bring the strays down to about one percent, and the library's sampler agrees with the loop to within 10^{-3}. The extra steps follow the curvature of the marginal flow.

17.2.3.2 The Reverse SDE and the Ancestral Chain

DDPM runs the reverse chain of Figure 17.5 instead, one stochastic step at a time. The reversal of a small forward step is nearly Gaussian (Sohl-Dickstein et al. 2015), and given the clean point it is exactly Gaussian: for $s < t$, $q(\mathbf{x}_s \mid \mathbf{x}_t, \mathbf{x}_0)$ has a mean linear in $\mathbf{x}_t$ and $\mathbf{x}_0$. Ho et al.'s *ancestral* step replaces $\mathbf{x}_0$ by $\hat{\mathbf{x}}_0$ in that mean and adds fresh noise with the variance $\beta = 1 - \bar{\alpha}_t/\bar{\alpha}_s$ of the forward step. On the VP path, written with the score, it reads

$$\mathbf{x}_s = \frac{\mathbf{x}_t + \beta\, \nabla \log p_t(\mathbf{x}_t)}{\sqrt{1-\beta}} + \sqrt{\beta}\,\mathbf{z}, \qquad \mathbf{z} \sim \mathcal{N}(\mathbf{0}, \mathbf{I}). \tag{17.24}$$

With $1/\sqrt{1-\beta} \approx 1 + \beta/2$, the step splits to first order into two:

$$\mathbf{x}_s \approx \Big[\mathbf{x}_t + \tfrac{\beta}{2}\nabla \log p_t(\mathbf{x}_t) + \sqrt{\beta}\,\mathbf{z}\Big] + \tfrac{\beta}{2}(\mathbf{x}_t + \nabla \log p_t(\mathbf{x}_t)).$$

The bracket is a Langevin step of size $h = \beta$, which leaves p_t in place and pulls stray points

back toward it, and the second term is one Euler step of the probability-flow ODE from t back to s, since on the VP path $\mathbf{u}_t = -\frac{1}{2}\beta(t)\,(\mathbf{x} + \nabla \log p_t)$ and $\beta \approx \beta(t)\,(t - s)$.

In continuous time the reversal is exact, by a theorem of Anderson (1982) . A forward process $d\mathbf{x} = \mathbf{f}\,dt + g\,d\mathbf{W}$ is reversed, from $t = 1$ down to 0, by

$$d\mathbf{x} = \left[\mathbf{f} - g^2 \nabla \log p_t(\mathbf{x})\right] dt + g\,d\bar{\mathbf{W}}, \qquad \text{which has the same marginals as} \qquad \frac{d\mathbf{x}}{dt} = \mathbf{f} - \tfrac{1}{2} g^2 \nabla \log p_t(\mathbf{x} \tag{17.25}$$

where $\bar{\mathbf{W}}$ is a Brownian motion run backward. For the VP path, $\mathbf{f} = -\frac{1}{2}\beta(t)\mathbf{x}$ and $g^2 = \beta(t)$, the right-hand ODE is the flow we just integrated, and the ancestral step is, to first order, the Euler–Maruyama step (Section 38.3.4.1) of the SDE. Between the two lies a family of samplers with the same marginals: the more noise a sampler injects, the larger the score correction in its drift (Section 38.4.5, Figure 38.20).

`ancestral_sample` takes this step on any path with any predictor `pred(x, alpha, sigma) -> (x0, eps)`, with the noise variance $\sigma_t^2 - (\alpha_t/\alpha_s)^2\sigma_s^2$ of the forward step from s to t.

```
@torch.no_grad()
def ancestral_sample(pred, x, path, steps, *, t_start=1.0):
    """Ho et al.'s sampler from x at t_start (1 is noise) to data at t = 0."""
    a, s = path.grid(steps, t_start)
    for i in range(steps):
        x0, eps = pred(x, a[i], s[i])
        ratio = s[i + 1] * a[i] / (a[i + 1] * s[i])
        x = a[i + 1] * x0 + s[i + 1] * ratio * eps      # the posterior mean
        if s[i + 1] > 0:                                # no noise at t = 0
            var = s[i] ** 2 - (a[i] / a[i + 1] * s[i + 1]) ** 2  # forward step
            x = x + math.sqrt(var) * torch.randn_like(x)
    return x
```

We run two hundred steps of the chain with the noise model, from the same $2{,}000$ starts as the flow.

```
pred_eps = make_pred(ema_eps, vp, 'eps')
print(mix.summary(ancestral_sample(pred_eps, x_T, vp, 200)))
```

```
{'weights': [0.515, 0.297, 0.189], 'stray': 0.0145, 'spread': 0.487}
```

The chain recovers the three modes with the right weights, up to the sampling error of $2{,}000$ points, and about as few strays as the flow at a hundred steps; Ho et al. (2020) ran a thousand steps, but in two dimensions two hundred suffice. Either sampler can use either network through `make_pred`.

17.2.4 Experiments on the Running Example

Neither network ever saw the exact posterior. We compare each with the oracle in its predictions, its samples, and through the dictionary.

17.2.4.1 The Learned Predictor against the Optimal One

The loss of any predictor splits into the floor and a gap that training can still reduce, $\mathbb{E}\|\epsilon - \hat{\epsilon}_\theta\|^2 = \mathbb{E}\|\epsilon - \epsilon^\star\|^2 + \mathbb{E}\|\hat{\epsilon}_\theta - \epsilon^\star\|^2$, with $\epsilon^\star$ the oracle's noise prediction. We evaluate all three for the noise model on $20{,}000$ fresh points at four times of the VP path.

```
def sq(u, w):
    """Mean squared distance between two batches of points."""
    return ((u - w) ** 2).sum(1).mean().item()

torch.manual_seed(4)
x0, eps = mix.sample(20_000), torch.randn(20_000, 2, device=device)
print('   t     gap   floor    loss')
with torch.no_grad():
    for t in (0.1, 0.3, 0.5, 0.7):
        tt = torch.full((len(x0),), t, device=device)
        x = vp.sample(x0, tt, eps)
        eps_hat = ema_eps(x, tt)
        eps_star = oracle(x, vp.alpha(tt), vp.sigma(tt))[1]
        print(f'{t:4} {sq(eps_hat, eps_star):7.4f} {sq(eps, eps_star):7.4f}'
              f' {sq(eps, eps_hat):7.4f}')
```

```
  t     gap   floor    loss
0.1  0.0004  1.3627  1.3625
0.3  0.0003  0.7730  0.7736
0.5  0.0001  0.4736  0.4736
0.7  0.0000  0.0540  0.0540
```

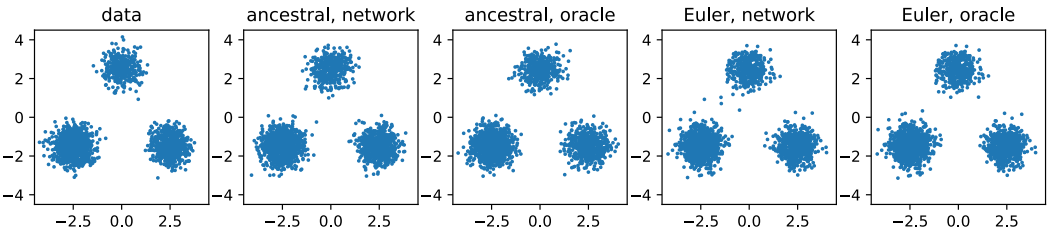
The loss column equals the floor plus the gap, to within the Monte Carlo error. The gap is below a thousandth at every time, on a target with unit variance per coordinate, and small next to the floor at every printed time: the network is close to the oracle wherever the data put their mass.

17.2.4.2 Samples from Both Samplers

A sampler accumulates the errors of all its steps. We draw $20{,}000$ samples with two hundred ancestral steps of the noise model and a hundred Euler steps of the velocity model, and repeat both with the oracle in place of the network. At this sample size a mode weight carries a standard error of about 0.0035.

```
torch.manual_seed(5)
x_T = torch.randn(20_000, 2, device=device)
runs = {'data': mix.sample(20_000),
        'ancestral, network': ancestral_sample(pred_eps, x_T, vp, 200),
        'ancestral, oracle': ancestral_sample(oracle, x_T, vp, 200),
        'Euler, network': ode_sample(pred_v, x_T, linear.sigma_bars(100)),
        'Euler, oracle': ode_sample(oracle, x_T, linear.sigma_bars(100))}
for name, x in runs.items():
    print(f'{name:>18}:', mix.summary(x))
d2l.scatter_panels([runs[k][:2000] for k in runs],
                   list(runs), lim=4.5, size=2.4);
```

```
             data: {'weights': [0.504, 0.299, 0.197], 'stray': 0.0103, 'spread': 0.476}
ancestral, network: {'weights': [0.503, 0.298, 0.199], 'stray': 0.0139, 'spread': 0.487}
 ancestral, oracle: {'weights': [0.503, 0.301, 0.196], 'stray': 0.011, 'spread': 0.483}
    Euler, network: {'weights': [0.5, 0.304, 0.197], 'stray': 0.0131, 'spread': 0.471}
     Euler, oracle: {'weights': [0.499, 0.304, 0.197], 'stray': 0.0107, 'spread': 0.474}
```



The ancestral chain with the network recovers the weights $0.5/0.3/0.2$ within sampling error, and Euler with the network matches them to within about 0.01. Every sampler, network or oracle, strays about as often as the data themselves, about one percent, the share of a Gaussian beyond three standard deviations. Both samplers recover the mode weights, which no single noise level could.

17.2.4.3 The Dictionary Checked

If both networks learned the posterior mean, each should match the linear-path velocity that Equation 17.21 computes from the exact score of p_t (`mix.score` with $\alpha = 1 - t$ and $\sigma = t$). For the noise model, `make_pred` returns $(\hat{\mathbf{x}}_0, \hat{\epsilon})$, whose difference is the linear-path velocity. We stop at $t = 0.95$, because a noise model has no clean-point estimate at $\alpha = 0$.

```
print('   t  E|u|^2  velocity net  noise net')
with torch.no_grad():
    for t in (0.2, 0.5, 0.8, 0.95):
        tt = torch.full((len(x0),), t, device=device)
        x = linear.sample(x0, tt, eps)
        u = -(x + t * mix.score(x, 1 - t, t)) / (1 - t)  # from the score
        x0_hat, eps_hat = pred_eps(x, 1 - t, t)
        print(f'{t:4} {sq(u, 0):7.2f} {sq(ema_v(x, tt), u):13.4f}'
              f' {sq(eps_hat - x0_hat, u):10.4f}')
```

```
   t  E|u|^2  velocity net  noise net
 0.2    8.05        0.0003     0.0005
 0.5    7.74        0.0007     0.0016
 0.8    1.07        0.0009     0.0017
0.95    2.00        0.0005     0.0136
```

The velocity network differs from the field computed from the score by less than a quarter of a percent of the field's mean squared norm, which ranges from about 8 at $t = 0.2$ to about 1 at $t = 0.8$. The noise network, converted through two paths and a change of time, stays within about one percent, with its largest error at $t = 0.95$, where the conversion divides by a small α. No score was learned, and no velocity appears in the noise model's loss.

17.2.5 Summary

A Gaussian path connects the data to noise through every noise level, and a walk down it settles the mode weights at coarse levels and the details at fine ones. One regression onto per-example targets trains one network along the whole path, because least squares recovers the posterior mean; the score, noise, clean point, v and velocity are affine functions of it, and the models of the literature differ only in path, target and sampler (Table 17.1). Samples come from the probability-flow ODE or the ancestral chain. On the mixture both learned models sit at their loss floors, agree through the dictionary, and recover the mode weights.

Table 17.1: The models of this chapter: the target and path that train the network, and the sampler that produces data.

model	target and path	sampler	where
denoising score model	the noise, at one level	Langevin dynamics	Section 17.1
noise-conditional score network	the noise at each level of a ladder, weighted by $\tilde{\sigma}^2$	annealed Langevin dynamics	here, 10
DDPM	ϵ on the VP path	the ancestral chain or the probability-flow ODE	here, Section 17.3
flow matching, rectified flow	$\epsilon - \mathbf{x}_0$ on the linear path	Euler steps on the probability-flow ODE	here, Section 17.4
v-prediction	**v** on the cosine path	either, through the dictionary	Section 17.5
masked diffusion	the clean tokens of a partly masked sequence, by a weighted cross-entropy	iterative unmasking	Section 17.8

Nothing in this section used the fact that $\mathbf{x}$ had two coordinates. An image has thousands, and its pixels are strongly correlated; the next question is what network, and what changes to training and sampling, a path through image space needs.

17.2.6 Exercises

1. **The marginal of the linear path.** The linear path Equation 17.14 joins a draw $\mathbf{x}_0$ of the mixture, with weights π_k, means μ_k and standard deviation $s = 0.5$ per coordinate, to independent noise ϵ.
   1. Show that p_t is the mixture with weights π_k, means $(1-t)\mu_k$ and per-coordinate variance $(1-t)^2 s^2 + t^2$. Derive Equation 17.21 from Equation 17.20.
   2. Show from Equation 17.17 that $\mathbf{u}_t(\mathbf{x}) \to \mathbf{x} - \mathbb{E}[\mathbf{x}_0]$ as $t \to 1$, and conclude where one Euler step from $t = 1$ to $t = 0$ sends every start.

3. With two Euler steps the first ends at $t = 1/2$. Show that it maps $\mathcal{N}(\mathbf{0}, \mathbf{I})$ to $\mathcal{N}(\mathbb{E}[\mathbf{x}_0]/2, \mathbf{I}/4)$, and compare the center and the width of this cloud with those of $p_{1/2}$. Which mode should the two-step samples overweight, and why?

2. **Velocity, noise and clean-point targets.** On the linear path, take the posterior mean of $\mathbf{x}_t = (1-t)\mathbf{x}_0 + t\epsilon$.
   1. Show that the marginal velocity equals both $(\mathbf{x} - \hat{\mathbf{x}}_0)/t$ and $(\hat{\epsilon} - \mathbf{x})/(1-t)$.
   2. By the first part, an error in $\hat{\mathbf{x}}_0$ reaches the velocity divided by t, and an error in $\hat{\epsilon}$ divided by $1-t$. Near which end of the path is each prediction a poor source of the velocity, and how does this explain why a noise model is never queried at $\alpha = 0$? (8 works out the propagation of errors between all targets on the cosine path.)
   3. Show that the velocity loss of $\mathbf{u}_\theta = (\mathbf{x} - \hat{\mathbf{x}}_{0,\theta})/t$ equals the clean-point loss $\mathbb{E}\|\hat{\mathbf{x}}_{0,\theta} - \mathbf{x}_0\|^2$ weighted by $1/t^2$, and the velocity loss of $\mathbf{u}_\theta = (\hat{\epsilon}_\theta - \mathbf{x})/(1-t)$ equals the noise loss weighted by $1/(1-t)^2$. Which times does flow matching emphasize relative to each of the two unweighted losses?

3. code **The loss floor.** Use `oracle` and `loss_floor`.
   1. Plot the floor of the velocity target on the linear path and of the noise target on the VP path against t on a grid in $(0, 1)$. Where is each largest? The printed training loss is a mean over $1{,}000$ minibatches of 512 points; estimate its standard error from `losses_v`, and decide whether a printed loss below the floor would be a contradiction.
   2. Let Σ be the covariance matrix of the mixture, with eigenvalues κ_1, κ_2. For Gaussian data with this covariance, show that the velocity floor on the linear path at time t is $\sum_i \kappa_i/((1-t)^2\kappa_i + t^2)$. Add this curve to the plot, and explain why it cannot lie below the floor of the mixture. Why do the two floors agree at both ends of the path?
   3. The score target $-\epsilon/\sigma_t$ has the floor of the noise target divided by σ_t^2. Evaluate it on the VP path at $t = 0.01$, 0.1 and 0.5, and explain why no network is trained on this target with t uniform on $(0, 1)$.
   4. The noise model's training loss is about a tenth of the velocity model's. Is it the better model? Wrap `ema_eps` and `ema_v` with `make_pred`, and at the noise levels of the linear path at $t \in \{0.1, 0.5, 0.9\}$ measure each network's gap to the oracle in a common unit, $\mathbb{E}\|\hat{\mathbf{x}}_0 - \hat{\mathbf{x}}_0^\star\|^2$, where $\hat{\mathbf{x}}_0^\star$ comes from `oracle`. Use the weights of Exercise 2c, and the factor α_t^2/σ_t^2 that turns a clean-point error into a noise error, to explain why the raw losses differ so much. Which comparison ranks the two models: the raw losses, or each loss's excess over its own floor?

   Part (d) is adapted from CMU 10-799, Homework 2 (Spring 2026), Question 3(a).

4. **The ancestral step to first order, for Gaussian data.** ▣ Let the data be $\mathcal{N}(\mathbf{0}, s^2\mathbf{I})$, so that on the DDPM chain $q_k = \mathcal{N}(\mathbf{0}, v_k\mathbf{I})$ with $v_k = \bar{\alpha}_k s^2 + 1 - \bar{\alpha}_k$.
   1. Show that one ancestral step Equation 17.24 with the exact score and step variance β_k maps q_k to $\mathcal{N}(\mathbf{0}, (v_{k-1} + \delta_k)\mathbf{I})$ with $\delta_k = \beta_k^2(1/v_k - 1)/(1-\beta_k)$.
   2. For which data is the step exact? Is the variance too large or too small for data narrower than the standard Gaussian, such as one component of the mixture, $s = 0.5$?
   3. How does the error accumulated over the chain change when the schedule is refined to twice as many steps with half the β_k?

5. **Step size and Gaussian reversal.** ▣ The ancestral chain uses Gaussian transitions, which are justified for small steps. This problem measures how small. Write $\sigma_{t|s}^2 = \sigma_t^2 - (\alpha_t/\alpha_s)^2\sigma_s^2$ for the variance of the forward step from s to $t > s$.

1. Show that the exact reversal is the mixture $q(\mathbf{x}_s \mid \mathbf{x}_t) = \mathbb{E}_{\mathbf{x}_0|\mathbf{x}_t}\big[q(\mathbf{x}_s \mid \mathbf{x}_t, \mathbf{x}_0)\big]$ and that its covariance is $\tau^2\mathbf{I} + c^2\,\mathrm{Cov}[\mathbf{x}_0 \mid \mathbf{x}_t]$, where $\tau^2 = \sigma_{t|s}^2\sigma_s^2/\sigma_t^2$ is the variance of $q(\mathbf{x}_s \mid \mathbf{x}_t, \mathbf{x}_0)$ and $c = \alpha_s\sigma_{t|s}^2/\sigma_t^2$ is the coefficient of $\mathbf{x}_0$ in its mean.
2. Let the data be ± 1 with equal probability in one dimension, and let $x_t = 0$. Show that the reversal is the equal mixture of $\mathcal{N}(\pm c, \tau^2)$, which is bimodal exactly when $\zeta^2 = c^2/\tau^2$ exceeds one, and that $\zeta^2 = \alpha_s^2\sigma_{t|s}^2/(\sigma_t^2\sigma_s^2)$. How does ζ shrink as the step $t - s$ is refined?
3. Evaluate ζ on the VP path from $t = 0.5$ to $s = 0.495$, 0.45 and 0.3, and for one step from $t = 0.3$ to $s = 0$. Which reversals are bimodal? What does a Gaussian transition with the optimal mean produce at $x_t = 0$ in the last case, and what should it produce?

6. code **Commitment time.** ▣ When along the path does a sample decide its mode? Biroli et al. (2024) measure this by cloning: start at a point of q_t, run two copies of the reverse chain with independent noise, and record whether they end in the same mode.
   1. Draw $2{,}000$ clean points, noise them to $\mathbf{x}_t$ on the VP path for $t = 0.9, 0.7, 0.5, 0.4, 0.3, 0.2, 0.1$, and run two copies of `ancestral_sample(pred_eps, x_t, vp, steps, t_start=t)` from each with about $200\,t$ steps. Plot the fraction $\varphi(t)$ of pairs whose ends have the same nearest mean. Which value does φ approach as $t \to 1$, and why?
   2. Let Λ be the largest eigenvalue of the covariance of the data. Biroli et al. define the speciation time by $\bar{\alpha}_t\Lambda = 1$, where the signal along the top principal direction stops dominating the noise. Compute it for the mixture and compare it with your curve and with the ladder of this section, whose coarse levels decided the mode weights. 3 and 6 work out the same question for two modes in one dimension.
7. **Same ratio, same flow.** ▣ Let two Gaussian paths have coefficients related by $\alpha_t^B = c_t\,\alpha_t^A$ and $\sigma_t^B = c_t\,\sigma_t^A$ for a smooth positive function c_t, so that they have the same log-SNR at every time.
   1. Show that $p_t^B(\mathbf{y}) = c_t^{-d}\,p_t^A(\mathbf{y}/c_t)$, and use Equation 17.20 to show that $\mathbf{u}_t^B(\mathbf{y}) = (\dot{c}_t/c_t)\,\mathbf{y} + c_t\,\mathbf{u}_t^A(\mathbf{y}/c_t)$. Conclude that the trajectories of the probability-flow ODE are related by $\mathbf{y}_t = c_t\,\mathbf{x}_t$.
   2. Let $\tau(t)$ be increasing. Show that the path $p_{\tau(t)}$ is generated by the field $\dot{\tau}(t)\,\mathbf{u}_{\tau(t)}$, with the same trajectories as $\mathbf{u}$.
   3. Find the change of time $\tau(t)$ for which the cosine path $(\cos\frac{\pi\tau}{2}, \sin\frac{\pi\tau}{2})$ at time $\tau(t)$ has the log-SNR of the linear path at time t, and show that $c_t = 1/\sqrt{(1-t)^2 + t^2}$. Where along the path is the rescaling largest?
   4. What does the probability-flow ODE of the cosine path do to data distributed as $\mathcal{N}(\mathbf{0}, \mathbf{I})$, and why? What does that of the linear path do to the same data?
8. **Ladder spacing in high dimensions.** ▣ Let the data be a single point at the origin of $\mathbb{R}^d$, so that $p_{\tilde{\sigma}} = \mathcal{N}(\mathbf{0}, \tilde{\sigma}^2\mathbf{I})$, and let the ladder be geometric with ratio $q = \tilde{\sigma}_{i-1}/\tilde{\sigma}_i > 1$.
   1. Show that $r = \|\mathbf{x}\|$ for $\mathbf{x} \sim p_{\tilde{\sigma}}$ satisfies $\mathbb{E}[r^2] = d\tilde{\sigma}^2$ and $\mathrm{Var}[r^2] = 2d\tilde{\sigma}^4$, and conclude that r is approximately $\mathcal{N}(\tilde{\sigma}\sqrt{d}, \tilde{\sigma}^2/2)$ for large d.
   2. A chain that leaves level $i-1$ with a sample of $p_{\tilde{\sigma}_{i-1}}$ enters level i at a radius that lies, with high probability, in $\tilde{\sigma}_{i-1}(\sqrt{d} \pm 3/\sqrt{2})$. Show that $p_{\tilde{\sigma}_i}$ puts the mass $\Phi(\sqrt{2d}(q-1) + 3q) - \Phi(\sqrt{2d}(q-1) - 3q)$ on this interval, where Φ is the standard normal distribution function. Why does a mass near zero mean that the chain starts the level where the network has seen almost no training inputs?

3. Y. Song and Ermon (2020) choose q so that this mass is about one half. Solve for q at $d = 3072$, the dimension of CIFAR-10 images, and count the levels from $\tilde{\sigma} = 50$ down to 0.01. Compare with the 232 levels of their CIFAR-10 model.
4. The top level must also be coarse enough to merge the modes. Two equal Gaussian components at $\pm\mu$ with variance v have a single mode exactly when $\mu^2 \le v$ (6). Use this criterion on the two lower modes of the mixture to estimate the noise level at which they merge, and compare it with the ladder's top level of 3.

9. code **The exact optimum memorizes.** A finite training set is itself a distribution, a mixture of point masses, and its exact velocity field is known. Bertrand et al. (2025) take as their starting point that integrating this field reproduces the training points, and ask why trained networks generalize instead.
   1. Draw 64 points from the mixture and train `TimeMLP(256, 3)` with `regression_loss(linear, 'velocity')` on batches drawn from these points only.
   2. `d2l.GaussianMixture(points, [1/64] * 64, 0.0)` is the empirical distribution, and its `velocity` method is the exact field for $t > 0$. Integrate it and the trained network with `euler` from the same $2{,}000$ starts. For both sets of samples, plot the histogram of the distance to the nearest training point.
   3. Which sampler reproduces the training set, and which produces new points? Bertrand et al. argue that the noise in the regression targets is not what makes the network generalize. Repeat the first two parts with $1{,}024$ training points, and describe how the two histograms change.
10. [extended] **A noise-conditional score network on the mixture.** ⊠⊠ Train the model of Song and Ermon (2019) on the running example and compare it with the samplers of this section.
    1. Train one `TimeMLP` on the ten levels of the ladder, with the level's log-noise $\log \tilde{\sigma}_i$ as its time input, by denoising score matching weighted by $\tilde{\sigma}_i^2$: the loss of level i is $\mathbb{E}\|\tilde{\sigma}_i\, \mathbf{s}_\theta(\mathbf{x}_0 + \tilde{\sigma}_i\epsilon, \tilde{\sigma}_i) + \epsilon\|^2$. Why does the weight make the levels comparable?
    2. Run annealed Langevin dynamics with the learned scores, 100 steps per level as in the text, and compare the summary with the exact-score ladder.
    3. The ladder used $1{,}000$ network evaluations. Compare it with `ancestral_sample` and `ode_sample` at the same number of evaluations, and with fewer. Which hand-chosen quantities of the ladder does the path make unnecessary?

17.3 Diffusion for Images

The model of the last section drew points in the plane, and nothing in it used the number two. The regression onto the velocity, the Euler steps that integrate it and the ancestral chain apply unchanged to a 28×28 image, a point in 784 dimensions. Most readers have used a system that turns a text prompt into a picture, and some have moved a slider that makes the picture follow the prompt more closely. What changes when $\mathbf{x}$ is an image? Four things: the network that maps a noisy image and a time to a prediction, the recipe that trains it, the way a label steers it, and the number that tells us whether its samples are good, now that "it looks right" has to be judged over thousands of images.

Each of the four has an origin. Ho et al. (2020) took the U-Net of Ronneberger et al. (2015), a segmentation network whose skip connections carry fine detail past a coarse bottleneck, and gave it the noise level as an extra input. Dhariwal and Nichol (2021) and then Ho and Salimans (2022) found that pushing each prediction further in the direction that a label adds makes samples more typical of the label and less varied; this *guidance* is what the prompt slider controls. Heusel et al. (2017) proposed to compare generated and real images through

the statistics of their features under a trained network, the measurement that image generation papers report.

Every image generator is built from these four parts, and the Image Models part scales each of them up. Here we make each one work on Fashion-MNIST, with a model that trains in a few minutes on one GPU.

```
torch.set_float32_matmul_precision('high')  # tf32 matmuls on tensor cores
```

17.3.1 The Denoising U-Net

17.3.1.1 Skips, Resolutions and the Time Embedding

The network of Section 17.2, `TimeMLP`, took a point and a time and returned a point of the same size. An image network keeps this contract: a noisy image $\mathbf{x}_t$ and the time t in, a prediction of the same shape out, here the velocity. Convolutions suit images, but a stack of 3×3 convolutions at full resolution makes each output pixel depend on a small window of the input, while at high noise the network must decide from the whole image whether a blurred blob is a shoe or a bag.

The U-Net works at several resolutions. An encoder halves the resolution twice, from 28×28 to 14×14 to 7×7, doubling the channels on the way, so that a 3×3 convolution at the bottom sees most of the image; a decoder upsamples back to 28×28. The bottleneck alone would lose fine detail, the edge of a sleeve or the stripes on a sneaker, so the decoder at each resolution also receives the encoder's features at that resolution, concatenated to its own. These *skip connections* give the network its U shape (Figure 17.7); the fully convolutional networks of Section 32.11 have the same encoder and decoder without them.

One network denoises at every noise level, and the right prediction depends strongly on the level, so the time enters every block. `d2l.time_embedding` of Section 17.1 maps t to sines and cosines at geometrically spaced frequencies, like the positional encodings of Section 10.4; a two-layer MLP turns these into one vector, and each block adds a learned projection of it to its feature maps, one bias per channel.

Each block applies two 3×3 convolutions with group normalization, adds a projection of the conditioning embedding between them, and keeps a residual connection around both. Group normalization computes its statistics over groups of channels within one image, so unlike batch normalization it does not mix the statistics of images at different noise levels.

```
class UNetBlock(nn.Module):
    """Two 3x3 convolutions with group norm, a residual, and an embedding."""
    def __init__(self, c_in, c_out, emb_dim=None):
        super().__init__()
        self.norm1 = nn.GroupNorm(8, c_in)
        self.conv1 = nn.Conv2d(c_in, c_out, 3, padding=1)
        self.emb = nn.Linear(emb_dim, c_out) if emb_dim else None
        self.norm2 = nn.GroupNorm(8, c_out)
        self.conv2 = nn.Conv2d(c_out, c_out, 3, padding=1)
        self.skip = (nn.Conv2d(c_in, c_out, 1) if c_in != c_out
                     else nn.Identity())
```

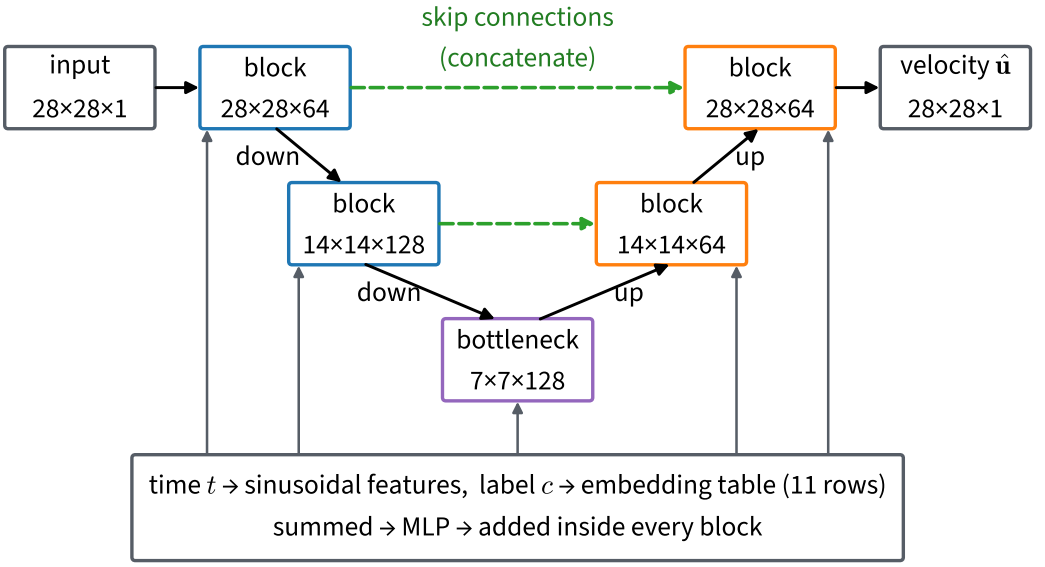


Figure 17.7: The U-Net of this section. Two down-sampling stages, a bottleneck at 7×7, and two up-sampling stages, with skip connections that concatenate encoder features to decoder features at the same resolution. The time and the class label are embedded, passed through a small MLP, and added inside every block.

```
    def forward(self, x, emb=None):
        h = self.conv1(F.silu(self.norm1(x)))
        if self.emb is not None:
            h = h + self.emb(emb)[:, :, None, None]
        h = self.conv2(F.silu(self.norm2(h)))
        return h + self.skip(x)
```

In the U-Net, two stride-2 convolutions go down to 7×7, and nearest-neighbor up-sampling to the size of each skip comes back.

```
class UNet(nn.Module):
    """A small U-Net on time t, a label y (num_classes = null), and cond."""
    def __init__(self, ch=64, num_classes=None, in_channels=1,
                 out_channels=1, emb_dim=128):
        super().__init__()
        self.emb_dim, self.num_classes = emb_dim, num_classes
        self.label_emb = (nn.Embedding(num_classes + 1, emb_dim)
                          if num_classes else None)
        self.emb_mlp = d2l.make_mlp([emb_dim] * 3, nn.SiLU())
        self.inc = nn.Conv2d(in_channels, ch, 3, padding=1)
        self.down1 = UNetBlock(ch, ch, emb_dim)                    # 28 x 28
        self.pool1 = nn.Conv2d(ch, ch, 3, stride=2, padding=1)     # 14 x 14
        self.down2 = UNetBlock(ch, 2 * ch, emb_dim)
        self.pool2 = nn.Conv2d(2 * ch, 2 * ch, 3, stride=2, padding=1)  # 7 x 7
```

```
        self.mid = UNetBlock(2 * ch, 2 * ch, emb_dim)
        self.up1 = UNetBlock(4 * ch, ch, emb_dim)                 # 14 x 14
        self.up0 = UNetBlock(2 * ch, ch, emb_dim)                 # 28 x 28
        self.outc = nn.Sequential(nn.GroupNorm(8, ch), nn.SiLU(),
                                  nn.Conv2d(ch, out_channels, 3, padding=1))

    def forward(self, x, t, y=None, cond=None):
        emb = d2l.time_embedding(t, self.emb_dim)
        if y is not None:
            emb = emb + self.label_emb(y)
        if cond is not None:
            emb = emb + cond
        emb = self.emb_mlp(emb)
        up = lambda h, skip: torch.cat([F.interpolate(
            h, size=skip.shape[-2:], mode='nearest-exact'), skip], 1)
        h1 = self.down1(self.inc(x), emb)
        h2 = self.down2(self.pool1(h1), emb)
        h = self.mid(self.pool2(h2), emb)
        h = self.up1(up(h, h2), emb)  # skip connection
        h = self.up0(up(h, h1), emb)  # skip connection
        return self.outc(h)
```

We build the network with 64 channels at full resolution and ten classes, and check that a batch of noisy images, times and labels keeps its shape.

```
torch.manual_seed(0)
dev = d2l.try_gpu()
net = UNet(ch=64, num_classes=10).to(dev)
x, t = torch.randn(8, 1, 28, 28, device=dev), torch.rand(8, device=dev)
c = torch.arange(8, device=dev)
print(f'output {tuple(net(x, t, c).shape)}, {d2l.num_params(net):,} parameters')
```

```
output (8, 1, 28, 28), 1,198,913 parameters
```

The network has about 1.2 million parameters, a thirtieth of the 35.7 million of the CIFAR-10 model of Ho et al. (2020).

17.3.1.2 Conditioning on a Class

A class label is one more input, and the U-Net already has a place for such inputs: the embedding that every block adds to its features. A table with one learned vector per class supplies the label's vector, which is added to the time features before the MLP. The table has one extra row, the *null label* $\emptyset$ (index 10 here), which stands for "no class given".

During training we replace each label by the null label with probability 0.1, a step called *label dropout*. The one network then learns two predictions: $\mathbf{u}_\theta(\mathbf{x}_t, t, c)$, the velocity of the images of class c, and $\mathbf{u}_\theta(\mathbf{x}_t, t, \emptyset)$, the velocity of all of Fashion-MNIST, trained on one example in ten. Ho and Salimans (2022) found dropout rates of 0.1 and 0.2 to work better than 0.5: the unconditional prediction needs only enough data to serve as a baseline for the conditional one (Section 17.3.4.2).

A sampler needs a field of $(\mathbf{x}, t)$ alone. `cfg` turns the conditional network into one: it evaluates the network with the requested labels c and with the null label in one doubled batch

and returns $\mathbf{u}_\emptyset + \gamma(\mathbf{u}_c - \mathbf{u}_\emptyset)$. With $\gamma = 1$ this is the conditional prediction $\mathbf{u}_c$, with $\gamma = 0$ the unconditional one; until the guidance experiment we use $\gamma = 1$.

```
def cfg(net, c, gamma):
    """Classifier-free guidance: (x, t) -> u + gamma (u_c - u), u unlabeled."""
    null = torch.full_like(c, net.num_classes)
    def guided(x, t):
        out_c, out = net(torch.cat([x, x]), torch.cat([t, t]),
                         torch.cat([c, null])).chunk(2)
        return out + gamma * (out_c - out)
    return guided
```

17.3.2 The Training Recipe

17.3.2.1 One Trainer

The loss is that of Section 17.2 with images in place of points: draw a clean image $\mathbf{x}_0$ in $[-1, 1]$ with its label, a time t uniform on $[0, 1]$ and noise ϵ, form $\mathbf{x}_t = (1 - t)\,\mathbf{x}_0 + t\,\epsilon$ on the linear path, and regress the network's output onto the conditional velocity $\epsilon - \mathbf{x}_0$. The only addition is label dropout, which `batch_fn` applies to each batch of 128 images.

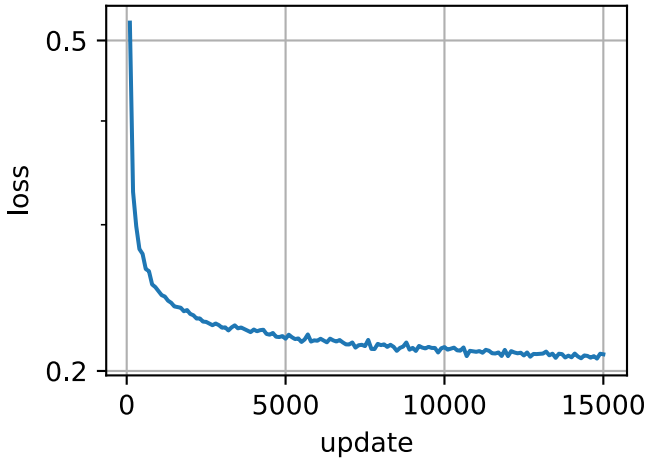
```
X, y, X_test, y_test = d2l.load_fashion_mnist_pm1()
path = d2l.GaussianPath.linear()

def batch_fn():
    i = torch.randint(0, len(X), (128,), device=X.device)
    drop = torch.rand(128, device=X.device) < 0.1   # label dropout
    return X[i], torch.where(drop, 10, y[i])        # 10 is the null label

def loss_fn(net, batch):
    x0, c = batch
    t, eps = torch.rand(len(x0), device=x0.device), torch.randn_like(x0)
    x_t = path.sample(x0, t, eps)
    return ((net(x_t, t, c) - path.target('velocity', x0, eps, t)) ** 2).mean()
```

`d2l.train_regression` of Section 17.1 runs Adam on this loss and returns the trained network, a moving average of its weights and the losses. We use a constant learning rate of 10^{-3} without warm-up for 15,000 updates, a few minutes on one GPU. Diffusion papers use a rate of $2 \cdot 10^{-4}$ after a warm-up (Ho et al. 2020), but for runs about fifty times longer than ours; at our budget the smaller rate learns too slowly, whichever target the network predicts.

```
num_steps = 15_000
net, ema, losses = d2l.train_regression(net, loss_fn, batch_fn, num_steps)
d2l.plot(torch.arange(100, num_steps + 1, 100),
         losses.reshape(-1, 100).mean(1).cpu(), 'update', 'loss', yscale='log')
```



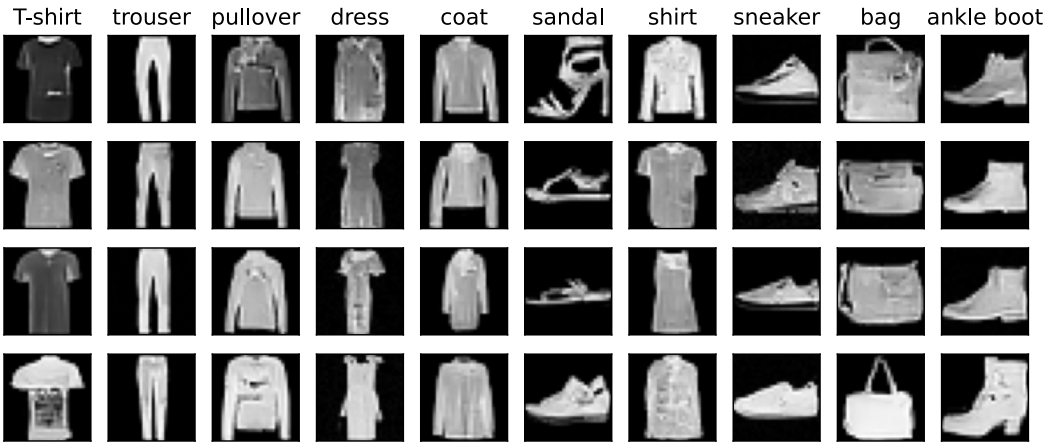
The loss, averaged over blocks of 100 updates, falls quickly over the first thousand updates and slowly after that. It levels off well above zero because its floor is the conditional variance of the velocity target (Section 17.2): at large t many clean images are consistent with one noisy image.

To sample, `d2l.make_pred` turns the network into a predictor of the clean image and the noise, and `d2l.ode_sample` integrates the probability-flow ODE from $t = 1$ to $t = 0$ with 50 Euler steps. `generate` draws n images, the i-th of class $i \bmod 10$, in chunks, starting from the same noise at every call so that two calls differ only in what we change between them.

```
euler50 = lambda pred, x: d2l.ode_sample(pred, x, path.sigma_bars(50))

def generate(model, gamma, sampler=euler50, n=5000, chunk=500):
    """n samples from fixed noise; the i-th requests class i % 10."""
    gen = torch.Generator(dev).manual_seed(0)
    z = torch.randn(n, 1, 28, 28, device=dev, generator=gen)
    c = torch.arange(n, device=z.device) % 10
    out = [sampler(d2l.make_pred(cfg(model, c_k, gamma), path, 'velocity'), z_k)
           for z_k, c_k in zip(z.split(chunk), c.split(chunk))]
    return torch.cat(out).clamp(-1, 1)

names = d2l.FashionMNIST().text_labels(range(10))
d2l.show_gray(generate(ema, 1.0, n=40), 4, 10, titles=names + [''] * 30);
```



Each column holds four samples of one class. Most are recognizable members of the requested class, with varied shapes, prints and shades; some are malformed, and shirts, pullovers and coats are hard to tell apart, as among real images.

17.3.2.2 The Moving Average

We sampled from `ema`, a moving average of the weights, not from the weights of the last update. With a constant learning rate the iterates of Adam do not settle: each update carries the noise of its minibatch, so the iterates wander in a cloud around a good region, and an average of recent iterates lies nearer its center (Polyak and Juditsky 1992). `d2l.EMA` of Section 16.5 keeps the exponential moving average

$$\bar{\theta} \leftarrow \delta\,\bar{\theta} + (1-\delta)\,\theta, \qquad \delta = 2^{-1/h}, \tag{17.26}$$

after every update, so that the weights of h updates ago count half as much as the current ones; h is the *half-life*. We use $h = 1000$.

The half-life has to suit the run. Too short, and the average keeps the noise of the last steps; too long, and it reaches back to early weights still far from the good region. Ho et al. (2020) use $\delta = 0.9999$, a half-life of about 7,000 updates, in a run of 800,000 updates; a fraction of that budget needs a proportionally shorter half-life.

17.3.3 Measuring Sample Quality

17.3.3.1 Fréchet Distance in Learned Features

A grid of forty samples shows whether samples look plausible. It cannot tell whether 5,000 of them have the right variety, whether there are too many bags and too few sandals, or which of two models that both look fine is better. For that we need a number that compares the distribution of the samples with that of real images.

Pixels are the wrong space for such a comparison: shifting a sneaker by two pixels moves the image far in Euclidean distance without changing what it shows. We therefore compare images in the feature space of a classifier, where images of similar content lie close together. The *Fréchet distance* (FD) of Heusel et al. (2017) fits Gaussians $\mathcal{N}(\mu_p, \Sigma_p)$ and $\mathcal{N}(\mu_q, \Sigma_q)$ to

the features of real and generated images and reports the squared Wasserstein-2 distance between them, derived in Section 16.5:

$$\mathrm{FD} = \|\mu_p - \mu_q\|^2 + \mathrm{tr}\Big(\Sigma_p + \Sigma_q - 2\big(\Sigma_p^{1/2}\Sigma_q\Sigma_p^{1/2}\big)^{1/2}\Big). \tag{17.27}$$

The first term grows when the average sample differs from the average real image, for example when one class is generated too often; the second when the spread differs, for example when all generated coats look alike. Both vanish only if the two Gaussians coincide.

Published scores, called FID, use the features of an Inception network trained on ImageNet. As in Section 16.5, we train our own instead: `d2l.FeatureCNN` (the online appendix "Utility Functions and Classes"), a small convolutional classifier with 128 features, trained on Fashion-MNIST for 15 epochs with the gray channel repeated three times. Because the feature network is our own, we call the score FD and never compare it with published values. `fashion_feature_cnn` trains it, and `fd_features` computes the features of images, 1,000 at a time; the network's head assigns each image a class.

```
def fashion_feature_cnn(X, y):
    """The feature network that judges Fashion-MNIST samples in this chapter."""
    cnn = d2l.FeatureCNN(resolution=28).to(X.device)
    d2l.train_feature_cnn(cnn, X.expand(-1, 3, -1, -1), y)
    return cnn

def fd_features(cnn, imgs):
    """Features of gray images (n, 1, 28, 28) in [-1, 1], on the CPU."""
    rgb = lambda x: x.clamp(-1, 1).expand(-1, 3, -1, -1)
    return d2l.batched(lambda x: cnn.features(rgb(x)), imgs).cpu()

cnn = fashion_feature_cnn(X, y)
f_test = fd_features(cnn, X_test)
c_test = cnn.head(f_test.to(dev)).argmax(1)
print(f'test accuracy {(c_test == y_test).float().mean():.3f}')
```

```
test accuracy 0.925
```

The feature network classifies more than 90% of the test images correctly, so its features separate the classes. `fd_and_agreement` computes the FD of a set of samples against reference features, together with the fraction of samples assigned to the requested class. We score 5,000 samples against the first 5,000 test images.

```
def fd_and_agreement(cnn, imgs, f_ref):
    """FD to the reference features, and the fraction of samples that the
    feature network assigns to the requested class i % 10."""
    f = fd_features(cnn, imgs)
    cls = cnn.head(f.to(imgs.device)).argmax(1)
    agree = (cls == torch.arange(len(imgs), device=cls.device) % 10)
    return d2l.fid_score(f, f_ref), agree.float().mean().item()

fd_ema, _ = fd_and_agreement(cnn, generate(ema, 1.0), f_test[:5000])
print(f'FD of 5,000 samples: {fd_ema:.2f}')
```

```
FD of 5,000 samples: 0.88
```

The samples score an FD of about 1. Whether that is small or large depends on the feature network and the number of images; the next subsection supplies a scale.

17.3.3.2 Reading the Number

Two sets of real images also have a positive FD: their sample means and covariances differ by chance, and the trace term of Equation 17.27 is biased upward, by roughly $1/n$ for n images per set. We therefore compute a *floor*, the FD between two disjoint sets of real test images of the size we score. Next to it we print the kernel inception distance (KID) of Bińkowski et al. (2018), an unbiased estimate of the squared maximum mean discrepancy with a cubic kernel on the same features (Section 16.5), whose expected value for two sets of real images is zero at every n.

```
for n in (1000, 5000):
    f_a, f_b = f_test[:n], f_test[5000:5000 + n]
    print(f'real against real, {n} images each: FD '
          f'{d2l.fid_score(f_a, f_b):.2f}, KID {d2l.kid_score(f_a, f_b):.5f}')
```

```
real against real, 1000 images each: FD 1.00, KID 0.02450
real against real, 5000 images each: FD 0.17, KID -0.00114
```

At 5,000 images per set the floor is about 0.2, and at 1,000 it is about 1, about six times larger. KID has no such bias: it lies within a few hundredths of zero at 1,000 images, and within a few thousandths at 5,000, which is also how far it scatters between other disjoint pairs of real sets. An FD can therefore be compared only with FDs from the same number of images, and the samples' FD of about 1 at 5,000 images is five to six times the floor: the gap is the error of our model after 15,000 updates.

With a scale in hand we can ask how much the moving average helps, and whether a thousand-step sampler beats fifty Euler steps. The reference is the ancestral chain of Section 17.2, run with 1,000 steps on the variance-preserving grid of Ho et al. (2020) through `make_pred`.

```
vp = d2l.GaussianPath.vp()
ancestral1000 = lambda pred, x: d2l.ancestral_sample(pred, x, vp, 1000)
fd_raw, _ = fd_and_agreement(cnn, generate(net, 1.0), f_test[:5000])
fd_anc, _ = fd_and_agreement(cnn, generate(ema, 1.0, ancestral1000),
                             f_test[:5000])
print(f'FD, Euler 50: average {fd_ema:.2f}, last weights {fd_raw:.2f}; '
      f'chain with 1,000 steps, average {fd_anc:.2f}')
```

```
FD, Euler 50: average 0.88, last weights 3.00; chain with 1,000 steps, average 0.70
```

The weights of the last update are no better than their average and often much worse: over training runs that differ only in the random seed, the average scored an FD of about 1 every time, the last weights anywhere from 1 to 3.5. The thousand-step chain scores a tenth to a quarter lower than fifty Euler steps, a modest gain for twenty times the evaluations. We measure with fifty Euler steps from now on.

17.3.4 Guidance

17.3.4.1 Bayes' Rule on Scores

Conditioning on a class looks like a new problem, but for scores it reduces to a sum. Bayes' rule at time t reads $p_t(\mathbf{x} \mid c) = p_t(\mathbf{x})\, p_t(c \mid \mathbf{x})/p(c)$, where $p_t(c \mid \mathbf{x})$ is the probability that a noisy image $\mathbf{x}$ at time t came from a clean image of class c, and $p(c)$ drops out of the gradient of the logarithm:

$$\nabla_{\mathbf{x}} \log p_t(\mathbf{x} \mid c) = \nabla_{\mathbf{x}} \log p_t(\mathbf{x}) + \nabla_{\mathbf{x}} \log p_t(c \mid \mathbf{x}). \tag{17.28}$$

The score of the class is the score of all the data plus the gradient of a classifier's log-probability. *Classifier guidance* (Dhariwal and Nichol 2021) supplies the second term with a separate classifier $p_\phi(c \mid \mathbf{x}, t)$ and multiplies it by a *guidance scale* γ:

$$\tilde{\mathbf{s}}(\mathbf{x}, t) = \nabla_{\mathbf{x}} \log p_t(\mathbf{x}) + \gamma\, \nabla_{\mathbf{x}} \log p_\phi(c \mid \mathbf{x}, t). \tag{17.29}$$

With $\gamma = 1$ this is Bayes' rule; Dhariwal and Nichol (2021) found that samples matched their classes much better with $\gamma > 1$. The classifier must be trained on noisy images, since the sampler evaluates its gradient at $\mathbf{x}_t$, where a classifier of clean images gives meaningless gradients. The appendix treats the rule on a one-dimensional mixture (Section 38.5.4); exercise 4 runs it on our images.

17.3.4.2 Classifier-Free Guidance

Classifier guidance needs a second network trained on noisy images, with its own errors. Ho and Salimans (2022) read Equation 17.28 from right to left instead:

$$\nabla_{\mathbf{x}} \log p_t(c \mid \mathbf{x}) = \nabla_{\mathbf{x}} \log p_t(\mathbf{x} \mid c) - \nabla_{\mathbf{x}} \log p_t(\mathbf{x}).$$

The classifier's gradient is the difference between the conditional and the unconditional score, and a network trained with label dropout provides both. Substituting its two scores $\mathbf{s}_c$ and $\mathbf{s}_\emptyset$ into Equation 17.29 gives the guided score $\tilde{\mathbf{s}} = \mathbf{s}_\emptyset + \gamma(\mathbf{s}_c - \mathbf{s}_\emptyset)$: classifier guidance with the classifier implicit in the network.

Our network predicts velocities, not scores, but at a fixed point $(\mathbf{x}, t)$ the velocity is an affine function of the score. On the linear path the dictionary of Section 17.2 gives

$$\mathbf{u}_t(\mathbf{x}) = -\frac{\mathbf{x} + t\, \nabla_{\mathbf{x}} \log p_t(\mathbf{x})}{1 - t}. \tag{17.30}$$

The term $-\mathbf{x}/(1-t)$ does not depend on the label, and the weights $1-\gamma$ and γ sum to one, so it passes through the combination unchanged, and the guided velocity is the same combination of the two velocities:

$$\tilde{\mathbf{u}}(\mathbf{x}, t) = \mathbf{u}_\theta(\mathbf{x}, t, \emptyset) + \gamma(\mathbf{u}_\theta(\mathbf{x}, t, c) - \mathbf{u}_\theta(\mathbf{x}, t, \emptyset)). \tag{17.31}$$

This is what `cfg` computes, and the same argument holds for every target of the dictionary (exercise 1). Ho and Salimans (2022) write the rule as $(1+w)\,\mathbf{u}_c - w\,\mathbf{u}_\emptyset$, so their w is our

$\gamma - 1$. Figure 17.8 draws the rule on the running example: for $\gamma > 1$ the guided prediction lies on the line through the unconditional and the conditional one, beyond the conditional one.

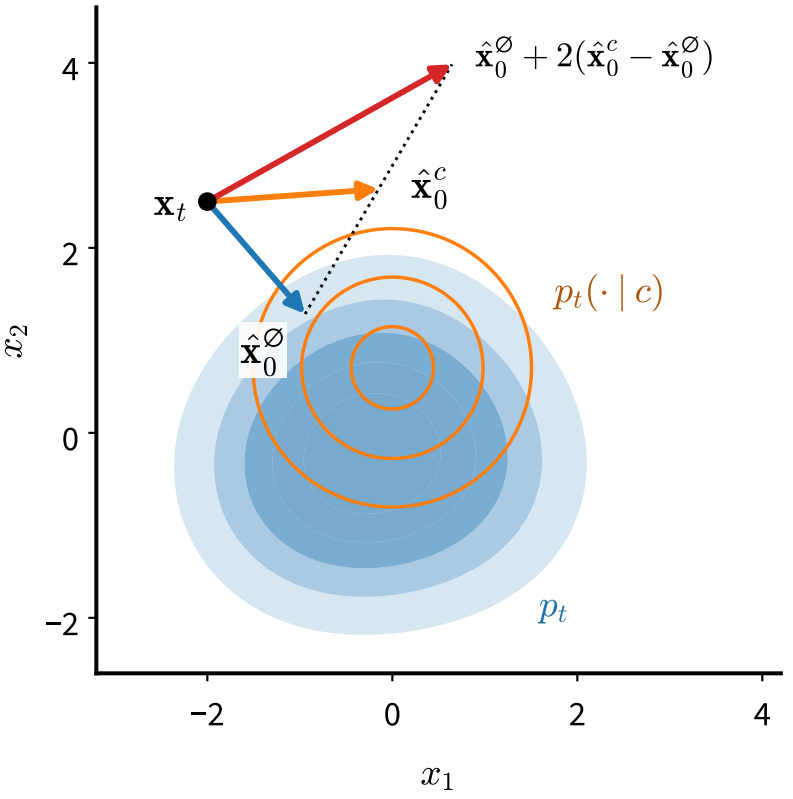


Figure 17.8: Classifier-free guidance on the three-mode mixture at one noise level. From a noisy point $\mathbf{x}_t$, arrows point to the posterior mean $\hat{\mathbf{x}}_0^{\emptyset}$ of all the data, to the posterior mean $\hat{\mathbf{x}}_0^{c}$ given the top mode, and to the guided prediction $\hat{\mathbf{x}}_0^{\emptyset} + \gamma(\hat{\mathbf{x}}_0^{c} - \hat{\mathbf{x}}_0^{\emptyset})$ at $\gamma = 2$, beyond $\hat{\mathbf{x}}_0^{c}$. Contours show the noised density and the noised density of the top mode.

What does γ do to the samples? We generate 5,000 samples at each of six scales from the same noise and report the FD, which measures how well the samples cover the distribution of real images, and the *class agreement*, the fraction of samples that the feature network assigns to the requested class.

```
for gamma in (0.0, 1.0, 1.5, 2.0, 3.0, 4.0):
    fd, agree = fd_and_agreement(cnn, generate(ema, gamma), f_test[:5000])
    print(f'gamma {gamma:.1f}: FD {fd:5.2f}, class agreement {agree:.3f}')
```

```
gamma 0.0: FD  3.10, class agreement 0.104
gamma 1.0: FD  0.88, class agreement 0.889
gamma 1.5: FD  1.03, class agreement 0.963
gamma 2.0: FD  1.84, class agreement 0.985
gamma 3.0: FD  3.65, class agreement 0.996
```

```
gamma 4.0: FD  5.25, class agreement 0.999
```

At $\gamma = 0$ the label plays no part, and the agreement is at chance, 10%. The FD of about 3 is more than three times that of the conditional model: the unconditional prediction covers all ten classes and was trained on only one example in ten. At $\gamma = 1$ the FD is about 1 and about 90% of the samples are assigned their requested class, about as high as this classifier can confirm, since it classifies only a little more than 90% of real test images correctly.

Guidance beyond $\gamma = 1$ trades variety for typicality. At $\gamma = 1.5$ the FD stays near its minimum, about 1, while the agreement rises to about 96%; the FDs at $\gamma = 1$ and 1.5 differ by less than the spread between training runs. At $\gamma = 2$ the agreement reaches about 98% and the FD has roughly doubled. At $\gamma = 3$ and 4 the agreement exceeds 99%, more than the feature network achieves on real images, while the FD climbs to about 3.5 and then to about 5: the guided samples are the class's prototypes, easier to classify than real ones, without the ambiguous examples that real data contain. The grids show the samples from the noise of the first grid at $\gamma = 0$ and at $\gamma = 4$.

```
for gamma in (0.0, 4.0):
    d2l.show_gray(generate(ema, gamma, n=40), 4, 10, titles=names + [''] * 30)
```



At $\gamma = 0$ the columns no longer follow their labels. At $\gamma = 4$ each image is a clear member of its class, but the four samples of a column resemble one another more than at $\gamma = 1$, with stronger contrast and fewer unusual shapes.

17.3.4.3 Guidance as a Tilt

Which distribution does the guided sampler follow? The guided score $\nabla \log p_t(\mathbf{x}) + \gamma \nabla \log p_t(c \mid \mathbf{x})$ is the gradient of the logarithm of the *tilted* density

$$\tilde{p}_t(\mathbf{x}) \propto p_t(\mathbf{x})\, p_t(c \mid \mathbf{x})^{\gamma} \propto p_t(\mathbf{x})^{1-\gamma}\, p_t(\mathbf{x} \mid c)^{\gamma}.$$

For $\gamma > 1$ the classifier's probability is raised to a power, which shifts mass toward the images about which the classifier is most certain; equivalently, part of the unconditional density is divided out, which pushes samples away from regions that the class shares with the others. This is the sharpening that the experiment measured.

The tilted densities are in general not the noised versions of any clean distribution. A density obtained by adding Gaussian noise of variance σ_t^2 cannot curve more sharply than the noise itself: no eigenvalue of $-\nabla^2 \log p_t$ exceeds $1/\sigma_t^2$. Raising a probability to a power $\gamma > 1$ can break this bound, and on the mixture of the appendix it does so at most noise levels for our scales (Section 38.5.4, exercise 8). The guided sampler therefore follows fields that no data distribution generates. Two remedies keep most of the gain in agreement at a smaller loss of variety: guidance only in a middle range of noise levels (Kynkäänniemi et al. 2024), and guidance by a weaker version of the model in place of the unconditional prediction (Karras, Aittala, Kynkäänniemi, et al. 2024); exercises 5 and 6 try both.

17.3.5 Summary

An image diffusion model is the model of Section 17.2 with a U-Net in place of the multilayer perceptron: an encoder and a decoder at three resolutions, skip connections that carry detail past the bottleneck, and the time and the label added inside every block. Trained for 15,000 updates and sampled from a moving average of its weights, a network of 1.2 million parameters produces recognizable Fashion-MNIST images. The Fréchet distance in the features of a trained classifier turns sample quality into a number, read against the floor of two sets of real images of the same size. Label dropout makes one network both conditional and unconditional, and classifier-free guidance extrapolates from the second through the first by a scale γ, trading variety, measured by the FD, for typicality, measured by class agreement.

Fifty Euler steps scored nearly as well as a thousand ancestral steps, at a twentieth of the cost. Which steps can be skipped, how few are enough, and which sampler should take them is the subject of Section 17.4.

17.3.6 Exercises

1. **Equivalent guidance rules.** Suppose that the network's two velocities are exact, so that, by the dictionary of Section 17.2, they are affine functions of the scores $\nabla \log p_t(\mathbf{x} \mid c)$ and $\nabla \log p_t(\mathbf{x})$, and that the classifier is the exact class posterior $p_t(c \mid \mathbf{x})$ of the noisy images.
    1. Show that classifier guidance Equation 17.29 at scale γ and classifier-free guidance Equation 17.31 at the same γ produce the same guided velocity.

   2. Every target of the dictionary (the noise, the clean image, the score, **v** and the velocity) is, at a fixed $(\mathbf{x}, t)$, an affine function $a(\mathbf{x}, t) + b(t)\,\hat{\mathbf{x}}_0$ of the posterior mean $\hat{\mathbf{x}}_0$. Show that applying the rule to any one target and converting gives the same guided prediction as applying it to any other. Which property of the weights $1-\gamma$ and γ does the argument use?
   3. Would the rule still be the same for every target if a network predicted $\hat{\mathbf{x}}_0$ clipped to $[-1, 1]$?
2. code **The floor of the Fréchet distance.** The floor printed in this section depends on the number n of images per set.
   1. Compute the FD and the KID between disjoint sets of test images for $n \in \{250, 500, 1000, 2500, 5000\}$, with as many disjoint pairs per n as the test set allows, up to five. Plot the FD against $1/n$. Is it close to a line, and where does the line meet $1/n = 0$?
   2. Compare the mean and the spread of the KID with those of the FD at each n. Which of the two would you report for a set of 500 samples?
   3. The samples of `generate(ema, 1.0)` scored an FD of about 1 at $n = 5000$. Predict their FD at $n = 1000$ from your line, and check the prediction.
   4. The FDs at $\gamma = 1$ and $\gamma = 1.5$ were within the spread between training runs. Score ten disjoint subsets of 500 samples at each of the two scales with both FD and KID. Can either score tell the two scales apart at this sample size?
3. code **What the Fréchet distance sees.** Corrupt the first 5,000 test images in controlled ways and score each corrupted set against the features of the other 5,000, `f_test[5000:]`: (i) remove one class and fill the set back to 5,000 with images of the other classes; (ii) double the share of one class at the expense of the others; (iii) blur every image with a Gaussian of standard deviation 0.5 or 1 pixel; (iv) add Gaussian noise of standard deviation 0.05; (v) replace 10% of the images by the mean image of their class.
   1. Tabulate the FD and the KID of each set next to the floor at 5,000 images. Which of these corruptions gives an FD of about 1, the score of our model?
   2. Is the FD more sensitive to a missing class or to a blur? What does the answer imply for reading the FD of a model as a measure of its variety or of the quality of its images?
4. code **A clean classifier as guide.** ⊠ Classifier guidance needs a classifier of noisy images, but one can also steer with a classifier of clean images applied to the network's estimate of the clean image.
   1. Report the accuracy of the feature network `cnn` on test images noised to $\mathbf{x}_t$ at $t \in \{0.25, 0.5, 0.75\}$ of the linear path. Why can its gradient at $\mathbf{x}_t$ not serve in Equation 17.29?
   2. Use the posterior mean $\hat{\mathbf{x}}_0(\mathbf{x}_t)$ of the unconditional prediction (`cfg(ema, c, 0.0)`) as the classifier's input instead, and add $\gamma \nabla_{\mathbf{x}_t} \log p_{\text{cnn}}(c \mid \hat{\mathbf{x}}_0(\mathbf{x}_t))$ to the unconditional score, converted to a velocity with the dictionary. Sample 1,000 images at $\gamma \in \{1, 3, 10\}$ with 50 Euler steps.
   3. Score the samples with `fd_and_agreement`. The class agreement is now measured by the same network that steered the sampler. Train a second `d2l.FeatureCNN` with another seed and score the samples with it as well. What does the comparison say about using a classifier both to guide and to evaluate?
5. code **Guidance in a limited interval.** Kynkäänniemi et al. (2024) apply guidance only within an interval of noise levels and use $\gamma = 1$ elsewhere. They measure the noise level by $\bar{\sigma}_t = \sigma_t/\alpha_t$ and found $0.28 < \bar{\sigma}_t \leq 2.9$ best for one of their ImageNet models.
   1. Find the times t of the linear path at which $\bar{\sigma}_t$ equals 0.28 and 2.9.

2. Write a version of `cfg` that applies the scale γ only for t in a given interval. At $\gamma = 4$, compute the FD and the class agreement of 5,000 samples for guidance in the high-noise interval, the middle interval and the low-noise interval, and compare with guidance at every step and with $\gamma = 1$.
3. Which interval accounts for most of the gain in agreement, and which for most of the rise in FD? Relate the answer to the time at which a sample commits to its class (exercise 6). 7 asks the same question for two classes in one dimension.

6. [extended] **Autoguidance.** ▯▯ Karras, Aittala, Kynkäänniemi, et al. (2024) guide a conditional model with a weaker version of itself in place of its unconditional prediction, $\tilde{\mathbf{u}} = \mathbf{u}_{\text{weak}}(\mathbf{x}, t, c) + \gamma(\mathbf{u}_\theta(\mathbf{x}, t, c) - \mathbf{u}_{\text{weak}}(\mathbf{x}, t, c))$.
   1. Treat each network as the exact velocity of its own densities $p_{\theta,t}(\mathbf{x} \mid c)$ and $p_{\text{weak},t}(\mathbf{x} \mid c)$. Show that the guided field is the velocity of a density proportional to $p_{\theta,t}(\mathbf{x} \mid c)\,(p_{\theta,t}(\mathbf{x} \mid c)/p_{\text{weak},t}(\mathbf{x} \mid c))^{\gamma-1}$, and write classifier-free guidance in the same form. Which images does each ratio favor? Use the answer to explain why classifier-free guidance reduces the variety within a class while autoguidance need not.
   2. Build two weak models: the moving average of the weights after 2,500 of the 15,000 updates, and a `UNet(ch=16, num_classes=10)` trained for 2,500 updates with the same recipe. Plot the FD against the class agreement for autoguidance with each weak model at $\gamma \in \{1.5, 2, 3, 4\}$, and for classifier-free guidance at the scales of the experiment.
   3. Which sampler reaches a given agreement at the lower FD, and which weak model guides better? Relate the answer to the requirement of Karras, Aittala, Kynkäänniemi, et al. (2024) that the weak model make the same kinds of errors as the main one.
7. code **Skip connections.** Write a variant of `UNet` without its two skip connections, in which `up1` and `up0` receive only the upsampled features, so that their input widths shrink to `2 * ch` and `ch`.
   1. Before training, predict at which times t the skip connections matter most and at which they hardly matter.
   2. Train both networks for 5,000 updates. For 1,000 test images, compute the squared error of the clean image implied by each network's velocity, $\hat{\mathbf{x}}_0 = \mathbf{x}_t - t\,\hat{\mathbf{u}}$, at $t \in \{0.1, 0.3, 0.5, 0.7, 0.9\}$, and plot both curves.
   3. Compare the FD of 5,000 samples of each network. Does the error curve of (b) predict the difference?
8. **Conditional scores of the mixture.** ▯ Let the data be the three-mode mixture of Section 17.1, with weights π_k, means μ_k and standard deviation s per coordinate, and let the label of a point be the component k that generated it. Add noise as in Section 17.1, $\tilde{\mathbf{x}} = \mathbf{x}_0 + \sigma\epsilon$, and write $v = s^2 + \sigma^2$.
   1. Show that the class posterior $p_\sigma(k \mid \tilde{\mathbf{x}})$ is a softmax over the logits $\ell_k(\tilde{\mathbf{x}}) = \log \pi_k + \mu_k^\top \tilde{\mathbf{x}}/v - \|\mu_k\|^2/(2v)$, which are affine in $\tilde{\mathbf{x}}$, and that the conditional score Equation 17.28 for label k is $(\mu_k - \tilde{\mathbf{x}})/v$.
   2. Merge the components with weights 0.3 and 0.2 into one class B. Write the gradient of $\log p_\sigma(B \mid \tilde{\mathbf{x}})$ in terms of the responsibilities of the components. Where in the plane is it small? In which proportions should samples of class B fall into its two components?
   3. A classifier fitted to clean points approximates the posterior at $\sigma = 0$. Show that its logits differ from those at level σ by the factor v/s^2 on every term except $\log \pi_k$, which at $\sigma = 3$ is 37. What does using its gradient at a coarse level amount to, in

terms of the tilted density of this section?

9. code **The rate of label dropout.** Train the conditional model with label dropout at rates 0.1, 0.2 and 0.5. For each, report the FD of 5,000 samples at $\gamma = 0$, and the FD and class agreement at $\gamma = 1$ and $\gamma = 1.5$. Which predictions improve and which degrade as the rate grows? Explain the result by how many updates each of the two predictions receives.
10. code **Diagnosing a broken trainer.** Each of four bugs leaves the code running: (i) the images are scaled to $[0, 1]$ instead of $[-1, 1]$ in `batch_fn`; (ii) `loss_fn` passes $1000\,t$ to the network, while the sampler passes t; (iii) `loss_fn` draws the noise once and reuses it for every minibatch; (iv) label dropout substitutes class 0 instead of the null label.
    1. Before running anything, predict for each bug how it shows in the training loss and in the samples.
    2. Inject each bug, train for 3,000 updates, and tabulate the final loss, the FD and the class agreement at $\gamma = 1.5$ next to those of the correct trainer at 3,000 updates. Where were your predictions wrong, and why?
    3. Which bugs leave the loss curve looking normal? Why can no training loss detect an error that lies in how the sampler uses the network?

    Adapted from CMU 10-799, Homework 1, Question 5, and Homework 2, Question 3 (Spring 2026).
11. code **Thresholding the predicted image.** ◪ The guided velocity implies a clean image $\hat{\mathbf{x}}_0 = \mathbf{x}_t - t\,\tilde{\mathbf{u}}$ on the linear path. Saharia et al. (2022) trace the saturated samples of large guidance scales to predictions $\hat{\mathbf{x}}_0$ outside the range of the data.
    1. For $\gamma \in \{1, 4, 16\}$, record at every Euler step the fraction of pixels of $\hat{\mathbf{x}}_0$ outside $[-1, 1]$ and plot it against t. Where along the path does the prediction leave the range, and how does γ change this?
    2. *Static thresholding* clips $\hat{\mathbf{x}}_0$ to $[-1, 1]$ before the step; *dynamic thresholding* sets, for each image, m to the larger of 1 and the 99.5th percentile of $|\hat{\mathbf{x}}_0|$, clips to $[-m, m]$ and divides by m. Rewrite the Euler step to form the velocity from a thresholded $\hat{\mathbf{x}}_0$, and implement both.
    3. Define the *saturation* of a set of images as the fraction of its pixels equal to 1 after clamping. Compare saturation, FD and class agreement of the three samplers at $\gamma \in \{4, 8, 16\}$.
12. **Guidance on Gaussian classes.** ◪ Let the images of class c be distributed as $\mathcal{N}(\mu_c, s^2\mathbf{I})$, and, for this sketch, model all images together as $\mathcal{N}(\mathbf{0}, S^2\mathbf{I})$ with $S^2 > s^2$. Both stay Gaussian on the linear path.
    1. Show that at $t = 0$ the guided score $\gamma \nabla \log p(\mathbf{x} \mid c) + (1 - \gamma)\nabla \log p(\mathbf{x})$ is the score of $\mathcal{N}(\kappa\mu_c, v\mathbf{I})$ with $v = s^2S^2/(\gamma S^2 - (\gamma - 1)s^2)$ and $\kappa = \gamma S^2/(\gamma S^2 - (\gamma - 1)s^2)$. For $\gamma > 1$, show that $v < s^2$ and $\kappa > 1$: guidance narrows the class and pushes it away from the mean of all images.
    2. In one dimension, with $\mu_c = 1$, $s^2 = 0.25$ and $S^2 = 2$, the guided velocity is affine in x at every t. Integrate the probability-flow ODE from $\mathcal{N}(0, 1)$ at $t = 0.999$ to $t = 0$ for $\gamma \in \{1, 2, 4\}$, tracking the mean and the variance, and compare them with those of (a). Show that for $\gamma > 1$ the guided Gaussians at $t > 0$ are not the noised versions of the Gaussian of (a), and explain why the ODE therefore does not end at it.
    3. The truncation trick of Exercise 5, applied to the generator $\mu_c + s\mathbf{z}$, and sampling from $p(\mathbf{x} \mid c)^{1/T}$ at a temperature $T < 1$ also narrow the class. Which of the three methods moves the class mean as well, and why?

17.4 Samplers, Inversion and Editing

Every step of a sampler costs one evaluation of the U-Net, and the ancestral chain of Section 17.2 takes a thousand steps per image. The last section sampled with fifty Euler steps of the probability-flow ODE, and on its conditional model the thousand-step chain scored only a little lower in FD, at twenty times the evaluations. Which of the thousand steps do we need, and how few can we take? And once sampling is the integration of a field, can we run it backwards, from a real image to the noise that produces it, and edit the image from there?

J. Song et al. (2021) answered the first question with an observation about training. The regression of Section 17.2 sees single noisy points $\mathbf{x}_t$ of training examples, never a chain of them, so it determines the distribution of $\mathbf{x}_t$ at each time and nothing more. Many samplers produce those distributions, among them a deterministic one that takes long strides and can be run in reverse, and one trained network serves all of them. Karras et al. (2022) read the deterministic sampler as a numerical method for a differential equation, separating two choices that earlier samplers had made together: the order of the method and the placement of its steps. C. Meng, He, et al. (2022) used the reverse direction to edit images.

We derive the family of samplers and its deterministic member, Euler's method, measure quality against evaluations on the mixture and on Fashion-MNIST, and end with inversion, editing and inpainting.

```
torch.set_float32_matmul_precision('high')  # tf32 matmuls on tensor cores
```

17.4.1 One Field, Two Samplers

The ancestral chain and Euler's method of Section 17.2, one adding fresh noise at every step and the other none, read the network through the same pair of estimates: at a point $\mathbf{x}_t = \alpha_t \mathbf{x}_0 + \sigma_t \epsilon$ of a Gaussian path, the clean point $\hat{\mathbf{x}}_0$ and the noise $\hat{\epsilon}$, each of which determines the other:

$$\hat{\mathbf{x}}_0 = \frac{\mathbf{x}_t - \sigma_t \hat{\epsilon}}{\alpha_t}. \tag{17.32}$$

Whatever the network was trained to predict, `d2l.make_pred` of Section 17.2 turns it into a function `pred(x, alpha, sigma)` returning this pair at any noise level, and the samplers of this section read nothing else.

17.4.1.1 The DDIM Family and Euler's Method

Take a time t and an earlier time $s < t$. A sampler step from t to s must turn a point distributed like $\mathbf{x}_t$ into one distributed like $\mathbf{x}_s$; the regression constrains nothing else. Write $\mathbf{x}_t = \alpha_t \mathbf{x}_0 + \sigma_t \epsilon$, choose a noise scale $\tilde{\sigma}$ between 0 and σ_s, and set

$$\mathbf{x}_s = \alpha_s \mathbf{x}_0 + \sqrt{\sigma_s^2 - \tilde{\sigma}^2}\, \epsilon + \tilde{\sigma}\, \xi, \qquad \xi \sim \mathcal{N}(\mathbf{0}, \mathbf{I}), \tag{17.33}$$

with ξ independent of ϵ. The two noise variances add to σ_s^2, so given $\mathbf{x}_0$ the point $\mathbf{x}_s$ has its correct distribution for every $\tilde{\sigma}$, and since $\epsilon = (\mathbf{x}_t - \alpha_t \mathbf{x}_0)/\sigma_t$, each $\tilde{\sigma}$ defines a step from $\mathbf{x}_t$ to $\mathbf{x}_s$. A sampler uses the network's estimates in place of $\mathbf{x}_0$ and ϵ:

$$\mathbf{x}_s = \alpha_s \hat{\mathbf{x}}_0 + \sqrt{\sigma_s^2 - \tilde{\sigma}^2}\,\hat{\epsilon} + \tilde{\sigma}\,\xi. \tag{17.34}$$

The step moves the predicted clean point to level s together with the predicted noise, and replaces part of that noise by fresh noise. J. Song et al. (2021) measure the fresh part by a strength $\eta \geq 0$,

$$\tilde{\sigma} = \eta\,\sigma_s\sqrt{1-\kappa^2}, \qquad \kappa = \frac{\sigma_s \alpha_t}{\alpha_s \sigma_t}. \tag{17.35}$$

Here $\kappa^2 = \mathrm{SNR}_t/\mathrm{SNR}_s < 1$, with $\mathrm{SNR} = \alpha^2/\sigma^2$. At $\eta = 1$, $\tilde{\sigma}^2$ is the variance of $\mathbf{x}_s$ given $\mathbf{x}_t$ and $\mathbf{x}_0$ under the noising chain of Section 17.2, and the step samples that chain's posterior with the estimates in place of the truth (exercise 2). At $\eta = 0$ no fresh noise enters and the step is deterministic: the *denoising diffusion implicit model* (DDIM), called implicit because it defines its samples by a map from noise rather than by a chain of densities.

`ddim_sample` runs the DDIM family on the grid of `path.grid` with any predictor `pred(x, alpha, sigma) -> (x0, eps)`, stopping at `t_end` (data by default) so that a loop can act on the sample between levels.

```
@torch.no_grad()
def ddim_sample(pred, x, path, steps, eta=0.0, *, t_start=1.0, t_end=0.0):
    """DDIM from x at t_start (1 is noise) to t_end; eta = 0 is deterministic.
    eta = 1 draws the posterior variance: not the ancestral sampler's."""
    a, s = path.grid(steps, t_start, t_end)
    for i in range(steps):
        x0, eps = pred(x, a[i], s[i])
        ratio = s[i + 1] * a[i] / (a[i + 1] * s[i])     # kappa of the text
        var = (eta * s[i + 1]) ** 2 * (1 - ratio ** 2)   # fresh noise
        x = a[i + 1] * x0 + math.sqrt(s[i + 1] ** 2 - var) * eps
        if var > 0:
            x = x + math.sqrt(var) * torch.randn_like(x)
    return x
```

How large an error does a long deterministic stride make? Take data that are themselves standard Gaussian, on the variance-preserving path. Then $\mathbf{x}_t$ is standard Gaussian at every time, so the exact sampler should move no point. The exact estimates are $\hat{\mathbf{x}}_0 = \alpha_t \mathbf{x}_t$ and $\hat{\epsilon} = \sigma_t \mathbf{x}_t$, and one deterministic stride from $t = 1$ to $t = 0$ returns $\hat{\mathbf{x}}_0 = \alpha_1 \mathbf{x}_1$: with $\alpha_1 \approx 0.0066$ the whole cloud collapses onto its mean. The step holds the noise estimate fixed across the stride, while along the exact trajectory it changes. In the coordinates $\bar{\mathbf{x}} = \mathbf{x}/\alpha_t$ and $\bar{\sigma}_t = \sigma_t/\alpha_t$ of Section 17.2, in which every Gaussian path reads $\bar{\mathbf{x}}_t = \mathbf{x}_0 + \bar{\sigma}_t \epsilon$, the deterministic step is $\bar{\mathbf{x}}_s = \hat{\mathbf{x}}_0 + \bar{\sigma}_s \hat{\epsilon}$, and substituting Equation 17.32 gives

$$\bar{\mathbf{x}}_s = \bar{\mathbf{x}}_t + (\bar{\sigma}_s - \bar{\sigma}_t)\,\hat{\epsilon},$$

one Euler step of the ODE $d\bar{\mathbf{x}}/d\bar{\sigma} = \hat{\epsilon}$. For the exact noise estimate this is the probability-flow ODE with the noise level $\bar{\sigma}$ as the clock, so on every Gaussian path DDIM is Euler's method. Section 38.5.4 proves this, together with the two extreme cases: a single data point is integrated without error by strides of any length, because $\hat{\epsilon}$ is then constant along each trajectory, and Gaussian data shrink at every stride (Figure 38.24). We check the equivalence

with the exact predictor of the mixture `mix` of Section 17.1, running DDIM and `ode_sample` for ten steps on the linear path from the same 20,000 starting points.

```
mix = d2l.GaussianMixture.three_modes()
vp, lin = d2l.GaussianPath.vp(), d2l.GaussianPath.linear()

def pred_exact(x, alpha, sigma):
    """The mixture's exact (x0, eps) at x = alpha x0 + sigma eps."""
    x0 = mix.posterior_mean(x, alpha, sigma)
    return x0, (x - alpha * x0) / sigma

torch.manual_seed(0)
z = torch.randn(20_000, 2)
gap = (d2l.ddim_sample(pred_exact, z, lin, 10)
       - d2l.ode_sample(pred_exact, z, lin.sigma_bars(10)))
print(f'DDIM against Euler: largest difference {gap.abs().max():.1e}')
```

```
DDIM against Euler: largest difference 4.8e-05
```

The two samplers agree to the rounding error of 32-bit arithmetic: they compute the same function in two coordinate systems.

17.4.1.2 Stochastic and Deterministic Steps

If one network serves every η, which strength should we use? With $f = \tilde{\sigma}^2/\sigma_s^2$, the step Equation 17.34 reads

$$\mathbf{x}_s = \alpha_s \hat{\mathbf{x}}_0 + \sigma_s\Big(\sqrt{1-f}\,\hat{\epsilon} + \sqrt{f}\,\xi\Big), \qquad f = \eta^2\Big(1 - \frac{\mathrm{SNR}_t}{\mathrm{SNR}_s}\Big). \tag{17.36}$$

Every member places the clean estimate at level s in the same way; a fraction f of the accompanying noise variance is fresh. On a fine grid f is small even at $\eta = 1$; on a grid of ten steps the $\eta = 1$ sampler replaces most of the noise at every step (exercise 1).

Fresh noise corrects errors. The probability-flow ODE is one member of a family of reverse processes with the same marginals (Section 38.4.5). A member that adds noise of variance h in a reverse step must also move by $\frac{h}{2}\nabla \log p_t$, and the added pair is the Langevin step of Section 17.1, which moves points toward the density it samples. A stochastic sampler therefore keeps pulling its state toward the model's distribution at each level, and an error made at one step is partly forgotten at the next. A deterministic step has no such pull: an error in $\hat{\mathbf{x}}_0$ moves the state, and the following steps carry it to the end.

The correction has a cost: a long stochastic step adds error of its own, because the exact reverse transition over a long step is far from Gaussian (exercise 5). We expect the deterministic sampler to win at small budgets and the stochastic one to catch up as the steps shorten. Figure 17.9 shows both on the mixture.

Two stochastic samplers must be kept apart. The ancestral chain of Section 17.2 adds the variance of the forward step, $\sigma_t^2 - (\alpha_t/\alpha_s)^2\sigma_s^2$, while $\eta = 1$ adds the posterior variance, smaller by the factor σ_s^2/σ_t^2. On a thousand-step grid the factor is close to one and the two nearly agree; on fifty steps the ancestral sampler adds more noise than the posterior allows.

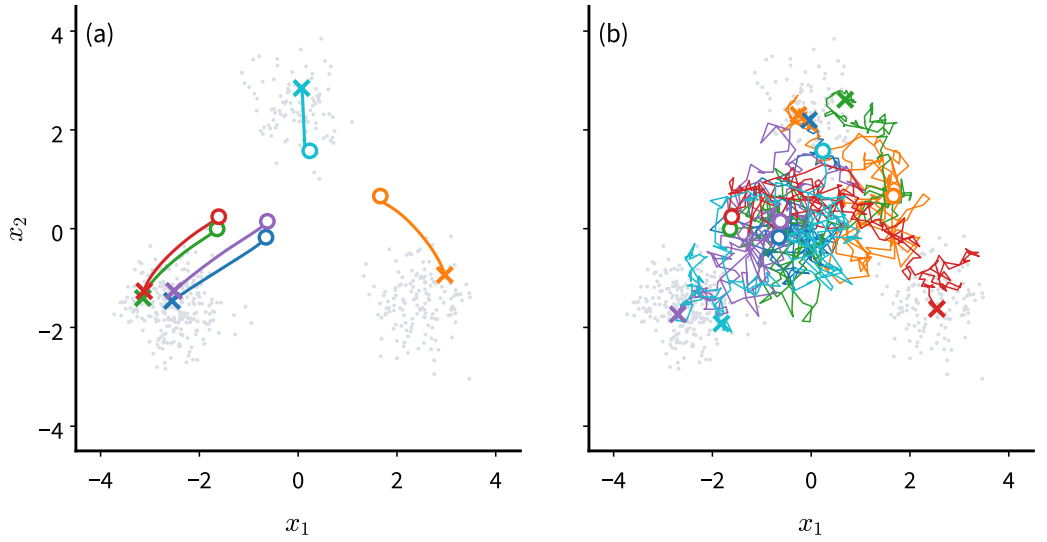


Figure 17.9: The two ends of the DDIM family on the three-mode mixture with the exact predictor, 200 steps on the variance-preserving path from six starting points at $t = 1$ (circles) to $t = 0$ (crosses); training data in grey. (a) At $\eta = 0$ each start follows a smooth curve into a mode. (b) At $\eta = 1$ the paths are jagged and may end in a different mode. Both produce the same distribution at $t = 0$.

17.4.2 Fewer Steps

We isolate the error of a stride on the mixture, then turn to second-order solvers and step grids, and measure quality against the number of evaluations on images.

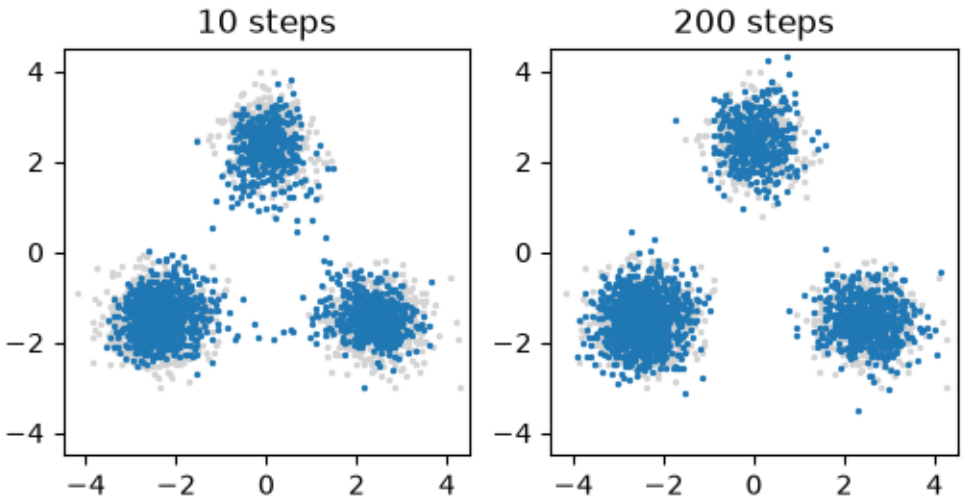
17.4.2.1 Skipping Steps on the Running Example

With the exact predictor, any difference between a ten-step sample and a thousand-step sample from the same start is error of the stride. The cell runs DDIM on the variance-preserving grid with 5 to 1,000 steps and prints the mean distance to the thousand-step samples and the spread (the mean squared distance to the nearest mode).

```
samples = {n: d2l.ddim_sample(pred_exact, z, vp, n)
           for n in (5, 10, 20, 50, 200, 1000)}
for n, x in samples.items():
    gap = (x - samples[1000]).norm(dim=1).mean()
    print(f'{n:4d} steps: distance to 1,000 steps {gap:.3f}, '
          f'spread {mix.summary(x)["spread"]:.3f}')
d2l.scatter_panels([samples[10][:2000], samples[200][:2000]],
                   ['10 steps', '200 steps'], data=mix.sample(2000));
```

```
  5 steps: distance to 1,000 steps 0.411, spread 0.309
 10 steps: distance to 1,000 steps 0.210, spread 0.380
 20 steps: distance to 1,000 steps 0.107, spread 0.420
 50 steps: distance to 1,000 steps 0.043, spread 0.451
200 steps: distance to 1,000 steps 0.009, spread 0.469
```

```
1000 steps: distance to 1,000 steps 0.000, spread 0.474
```



The distance halves each time the number of steps doubles, as for a first-order method: about 0.4 at five steps, 0.2 at ten and 0.04 at fifty, on a scale where the modes are five units apart. The error has a direction: the spread falls from about 0.47 at a thousand steps to 0.38 at ten, and the clouds in the left panel are visibly tighter than the data behind them. A stride holds the noise estimate at its value at the start, which for Gaussian data always removes too much noise (exercise 3), and the modes of the mixture behave like Gaussians. Since the spread rewards contraction, and a stochastic sample has no thousand-step counterpart, we compare $\eta = 0$ with $\eta = 1$ on images instead.

17.4.2.2 Second Order and the Step Grid

Euler's method holds the slope fixed across a step, so its error over the whole run falls only in proportion to the step size. Heun's method (Section 38.1.3) takes the Euler step as a trial, evaluates the slope again at the trial point, and moves with the average of the two slopes:

$$\bar{\mathbf{x}}' = \bar{\mathbf{x}}_t + (\bar{\sigma}_s - \bar{\sigma}_t)\,\hat{\epsilon}(\bar{\mathbf{x}}_t, \bar{\sigma}_t), \qquad \bar{\mathbf{x}}_s = \bar{\mathbf{x}}_t + (\bar{\sigma}_s - \bar{\sigma}_t)\,\frac{\hat{\epsilon}(\bar{\mathbf{x}}_t, \bar{\sigma}_t) + \hat{\epsilon}(\bar{\mathbf{x}}', \bar{\sigma}_s)}{2}. \tag{17.37}$$

Its error falls with the square of the step size, at two evaluations per step. For standard Gaussian data the noise estimate is $\hat{\epsilon} = \bar{\sigma}\bar{\mathbf{x}}/(1 + \bar{\sigma}^2)$, and the exact flow multiplies $\bar{\mathbf{x}}$ by $\sqrt{(1 + \bar{\sigma}_s^2)/(1 + \bar{\sigma}_t^2)}$. From $\bar{\sigma}_t = 1$ to $\bar{\sigma}_s = 0.5$ the exact factor is 0.791, Euler's step gives 0.75 and Heun's 0.80: errors of 0.041 and 0.009, a fifth of the error for twice the cost. The last step, to $\bar{\sigma} = 0$, is an Euler step, and so is a first step from $\bar{\sigma} = \infty$, so `heun_sample` spends $2n - 1$ evaluations on n steps, or $2n - 2$ from pure noise on the linear path.

Whether the second evaluation pays depends on where the steps are, and the answer depends on the field. On the exact field of a one-dimensional mixture (Section 38.6.4), Heun's method loses to Euler's below about forty evaluations when the steps are equal in t, and wins at every budget from ten evaluations when the steps are crowded toward the data: on

equal steps Euler's step errors of opposite sign largely cancel while Heun's do not, an accident that crowding the steps removes. Karras et al. (2022) crowd the steps by spacing $\bar{\sigma}^{1/\rho}$ evenly,

$$\bar{\sigma}_i = \left(\bar{\sigma}_{\max}^{1/\rho} + \tfrac{i}{n-1}\left(\bar{\sigma}_{\min}^{1/\rho} - \bar{\sigma}_{\max}^{1/\rho}\right)\right)^{\rho}, \qquad i = 0, \dots, n-1, \tag{17.38}$$

followed by a last step to $\bar{\sigma} = 0$, with $\bar{\sigma}_{\max} = 80$, $\bar{\sigma}_{\min} = 0.002$ and $\rho = 7$ for images in $[-1, 1]$; this is `edm_grid`. The grid is a choice of the sampler, separate from the training path, since `make_pred` translates any level to the network's own time. The cell saves `edm_grid` and `heun_sample`, and prints $\bar{\sigma}$ on this grid and on the ten-step grids of the variance-preserving and linear paths, even in time.

```
def edm_grid(steps, sigma_max=80.0, sigma_min=0.002, rho=7):
    """Karras et al.'s levels sigma_bar: even in sigma_bar^(1/rho), then 0."""
    hi, lo = sigma_max ** (1 / rho), sigma_min ** (1 / rho)
    return [(hi + i / (steps - 1) * (lo - hi)) ** rho
            for i in range(steps)] + [0.0]

@torch.no_grad()
def heun_sample(pred, x, sigma_bars):
    """Heun on d x_bar / d sigma_bar = eps: an Euler trial, then the average
    of the two slopes; from x at the first level to data, as ode_sample."""
    r0 = sigma_bars[0]
    xb = x if math.isinf(r0) else x * math.sqrt(1 + r0 ** 2)  # x / alpha
    for r_t, r_s in zip(sigma_bars[:-1], sigma_bars[1:]):
        x0, eps = d2l.pred_bar(pred, xb, r_t)
        x_next = x0 + r_s * eps                       # the Euler step
        if r_s > 0 and not math.isinf(r_t):           # no slope at 0 or inf
            x_next = xb + (r_s - r_t) * (
                eps + d2l.pred_bar(pred, x_next, r_s)[1]) / 2
        xb = x_next
    return xb

for name, levels in (('VP', vp.sigma_bars(10)), ('linear', lin.sigma_bars(10)),
                     ('EDM', d2l.edm_grid(10))):
    print(f'{name:7s}', ' '.join(f'{r:7.3g}' for r in levels))
```

```
VP          152    58.8    25.1    11.8     6.1    3.41    2.03    1.23    0.72    0.34       0
linear      inf       9       4    2.33     1.5       1   0.667   0.429    0.25   0.111       0
EDM          80    42.4    21.1    9.72    4.07     1.5    0.47   0.117  0.0204   0.002       0
```

The variance-preserving grid spends four of its ten steps above $\bar{\sigma} = 6$, where all noisy points look alike, and reaches the data from 0.34. The linear grid's finite levels run from 9 to 0.11. The EDM grid starts at 80 but crowds its last levels down to 0.002. Exponential integrators such as DPM-Solver (Lu, Zhou, et al. 2022) take steps in $\log \bar{\sigma}$ (exercise 6).

17.4.2.3 Quality against Evaluations on Images

On images the field is learned. We train the unconditional version of the model of Section 17.3, the U-Net of about 1.2 million parameters without its label input, on the velocity

of the linear path for ten thousand updates, a few minutes on one GPU, and sample from the moving average of its weights.

```
X, y, X_test, y_test = d2l.load_fashion_mnist_pm1()

def batch_fn():
    return X[torch.randint(0, len(X), (128,), device=X.device)]

torch.manual_seed(0)
net = d2l.UNet(ch=64).to(X.device)
net, ema, _ = d2l.train_regression(net, d2l.regression_loss(lin, 'velocity'),
                                   batch_fn, 10_000)
pred = d2l.make_pred(ema, lin, 'velocity')
```

We score samples by the Fréchet distance (FD) of Section 17.3 in `d2l.FeatureCNN` features, 1,000 generated against 1,000 test images, read against the floor of two disjoint sets of 1,000 real images.

```
cnn = d2l.fashion_feature_cnn(X, y)
features = lambda imgs: d2l.fd_features(cnn, imgs)
f_test = features(X_test[:1000])
floor = d2l.fid_score(features(X_test[1000:2000]), f_test)
print(f'floor, 1,000 real images against 1,000 others: FD {floor:.2f}')
```

```
floor, 1,000 real images against 1,000 others: FD 0.80
```

Each sampler draws 1,000 images from the same noise. “VP” and “linear” denote even steps in time on that path, so DDIM with $\eta = 0$ on “VP” is Euler’s method on the first grid printed above; “EDM” denotes the crowded grid. We count network evaluations, not steps.

```
families = {  # name: (sampler(pred, noise, n), step counts n)
    'ancestral, VP': (lambda p, z, n: d2l.ancestral_sample(p, z, vp, n),
                      (50, 200, 1000)),
    'DDIM eta=1, VP': (lambda p, z, n: d2l.ddim_sample(p, z, vp, n, 1.0),
                       (10, 20, 50, 100)),
    'DDIM eta=0, VP': (lambda p, z, n: d2l.ddim_sample(p, z, vp, n),
                       (10, 20, 50, 100)),
    'Euler, linear': (lambda p, z, n: d2l.ode_sample(p, z, lin.sigma_bars(n)),
                      (10, 20, 50)),
    'Euler, EDM': (lambda p, z, n: d2l.ode_sample(p, z, d2l.edm_grid(n)),
                   (10, 20, 50)),
    'Heun, linear': (lambda p, z, n: d2l.heun_sample(p, z, lin.sigma_bars(n)),
                     (5, 10, 25)),
    'Heun, EDM': (lambda p, z, n: d2l.heun_sample(p, z, d2l.edm_grid(n)),
                  (5, 10, 25))}

def evaluations(name, n):  # Heun's second slope is skipped at inf and at 0
    return {'Heun, linear': 2 * n - 2, 'Heun, EDM': 2 * n - 1}.get(name, n)

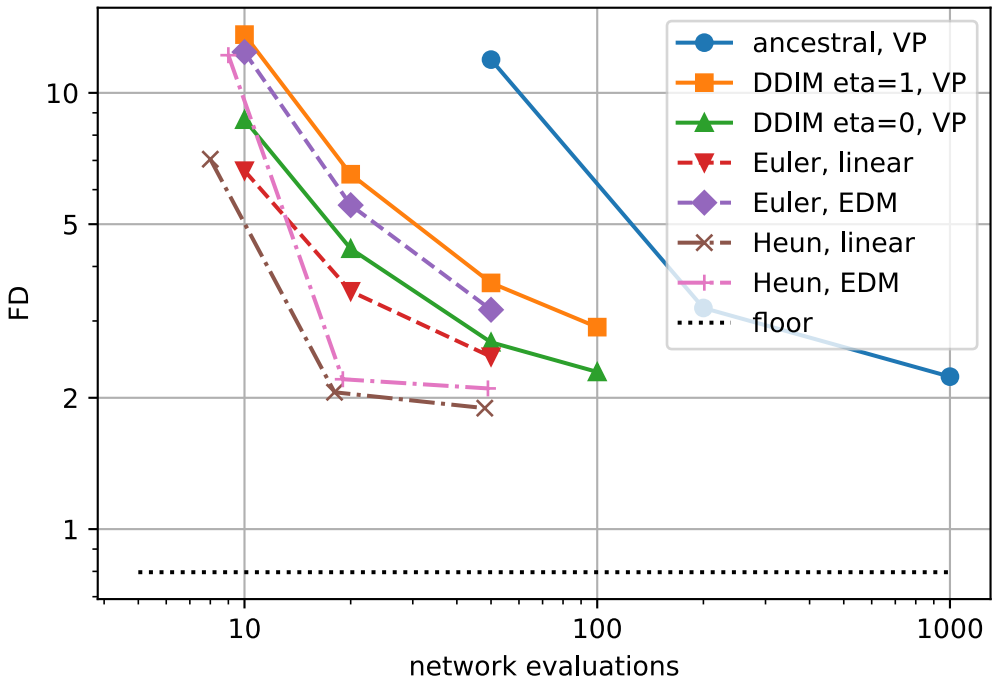
torch.manual_seed(1)
noise = torch.randn(1000, 1, 28, 28, device=X.device)
```

```
images = {name: [f(pred, noise, n).clamp(-1, 1) for n in steps]
          for name, (f, steps) in families.items()}
fd = {name: [d2l.fid_score(features(x), f_test) for x in xs]
      for name, xs in images.items()}
for name, (_, steps) in families.items():
    print(f'{name:15s}', '  '.join(f'{evaluations(name, n):4d}: {v:5.2f}'
                                   for n, v in zip(steps, fd[name])))
```

```
ancestral, VP     50: 11.92   200:  3.21  1000:  2.24
DDIM eta=1, VP    10: 13.60    20:  6.51    50:  3.67   100:  2.90
DDIM eta=0, VP    10:  8.68    20:  4.39    50:  2.68   100:  2.29
Euler, linear     10:  6.63    20:  3.51    50:  2.48
Euler, EDM        10: 12.40    20:  5.53    50:  3.18
Heun, linear       8:  7.05    18:  2.06    48:  1.89
Heun, EDM          9: 12.20    19:  2.21    49:  2.10
```

The plot shows each FD against the number of evaluations.

```
evals = [[evaluations(name, n) for n in steps]
         for name, (_, steps) in families.items()]
d2l.plot(evals + [[5, 1000]], list(fd.values()) + [[floor, floor]],
         'network evaluations', 'FD', legend=list(families) + ['floor'],
         xscale='log', yscale='log', figsize=(6, 4),
         fmts=('o-', 's-', '^-', 'v--', 'D--', 'x-.', '+-.', 'k:'))
```



The thousand-step ancestral chain, the reference, scores two to three times the floor. Deterministic DDIM at fifty steps scores about 3, within about 1.5 times the chain's FD at a twentieth of the evaluations, which reproduces the central result of J. Song et al. (2021) .

At equal numbers of steps, $\eta = 0$ beats $\eta = 1$ by about a third at 10, 20 and 50 steps, and the gap narrows to about a fifth at a hundred, as the trade-off above predicts. The ancestral sampler at fifty steps scores worse than deterministic DDIM at ten, because it adds more noise than the posterior allows; its images are speckled (exercise 7).

Heun's method on the crowded grid scores about 2 at nineteen evaluations, better than DDIM at fifty; on the linear grid at forty-eight evaluations it scores about 2, the quality of the thousand-step chain at a twentieth of its evaluations. The grid matters as much as the method: with ten steps, Euler's method scores about 7 on the linear grid, 9 to 10 on the variance-preserving grid (where it is DDIM), and about 13 on the crowded grid. Crowding the steps toward the data hurts Euler's method at every budget, and Heun's at eight and at forty-eight evaluations; only near twenty do the two grids tie. On this model the even grid of the linear path serves both methods best.

The one-dimensional result does not transfer, and the printed grids suggest why. With ten levels the crowded grid spends four steps above $\bar{\sigma} = 4$, where all noisy images look alike, and three below 0.12, where a 28×28 garment barely changes, leaving three steps for the range in which its shape is decided; the linear grid spends seven of its ten steps there. Karras et al. (2022) tuned $\rho = 7$ for 35 to 79 evaluations on larger images.

Below are eight images from each of four samplers, all from the same eight noise images: the ancestral chain with 1,000 steps, DDIM with 10 and with 50 steps, and Heun's method on the crowded grid with 19 evaluations.

```
rows = [images['ancestral, VP'][2], images['DDIM eta=0, VP'][0],
        images['DDIM eta=0, VP'][2], images['Heun, EDM'][1]]
d2l.show_gray(torch.cat([x[:8] for x in rows]), 4, 8);
```



The last three rows match column by column: the deterministic samplers approximate one map from noise to images, and ten steps slightly blur the image that fifty steps and Heun's method agree on. The first row does not match, because the ancestral chain's image depends as much on the noise it adds as on its start.

17.4.3 Inversion and Editing

A deterministic sampler integrates a differential equation, which can be run in either direction. Running it backwards assigns each image a noise image, its *code*, which gives interpolation, editing and inpainting without retraining.

17.4.3.1 The Latent Code

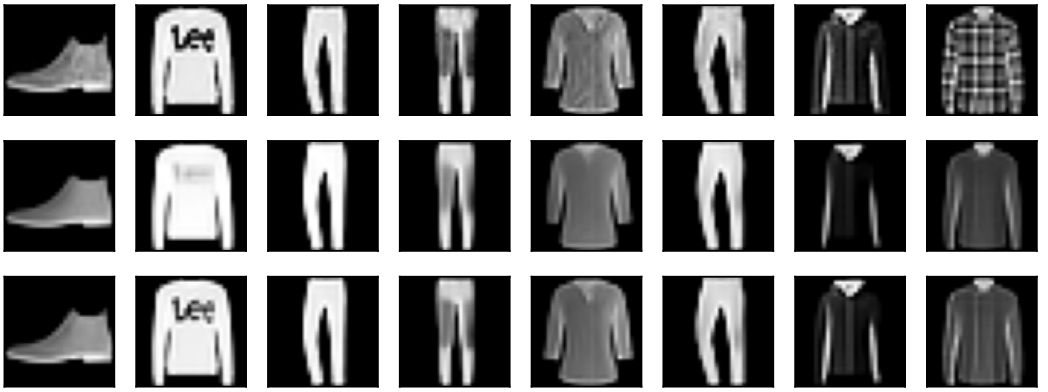
`ddim_invert` runs the deterministic sampler backwards, from an image at $t = 0$ toward noise. Each step evaluates the network at the current point but tells it the noise level of the step's upper end, since the predictor needs $\sigma > 0$ and the first point has none.

```
@torch.no_grad()
def ddim_invert(pred, x, path, steps, t_end=1.0):
    """Deterministic DDIM run backwards, from data x at t = 0 to t_end."""
    a, s = path.grid(steps, 0.0, t_end)
    for i in range(steps):
        eps = pred(x, a[i + 1], s[i + 1])[1]    # x read at the next level
        x = a[i + 1] * (x - s[i] * eps) / a[i] + s[i + 1] * eps
    return x
```

In the coordinates $(\bar{\mathbf{x}}, \bar{\sigma})$ each inverse step is the Euler step $\bar{\mathbf{x}}_{t'} = \bar{\mathbf{x}}_t + (\bar{\sigma}_{t'} - \bar{\sigma}_t)\,\hat{\epsilon}$ taken upward, from the level t to the next level $t' > t$. The decoder evaluates the noise estimate at the upper point, the encoder at the lower one, the only point it has; the two differ by the change of $\hat{\epsilon}$ over one step, so the round trip is inexact even with an exact predictor. The cell encodes and decodes 100 test images with 10 to 100 steps on the linear path.

```
x0 = X_test[:100]
codes = {n: d2l.ddim_invert(pred, x0, lin, n) for n in (10, 20, 50, 100)}
recon = {n: d2l.ddim_sample(pred, code, lin, n) for n, code in codes.items()}
for n in codes:
    print(f'{n:3d} steps: squared error per pixel '
          f'{((recon[n] - x0) ** 2).mean():.4f}, code std {codes[n].std():.2f}')
d2l.show_gray(torch.cat([x0[:8], recon[10][:8], recon[50][:8]]), 3, 8);
```

```
 10 steps: squared error per pixel 0.0610, code std 0.55
 20 steps: squared error per pixel 0.0390, code std 0.61
 50 steps: squared error per pixel 0.0175, code std 0.73
100 steps: squared error per pixel 0.0099, code std 0.80
```



The reconstruction error falls with the number of steps, more slowly than the step size, since the learned field adds errors of its own: at fifty steps (third row) the reconstructions are close to the originals, at ten (second row) they lose fine detail. The codes are not typical noise: their standard deviation lies below one, approaching one only as the steps get shorter.

Because the sampler is a continuous function of its code, we can move between two images by interpolating between their codes. A straight line is the wrong path: the average of two independent noise images has a norm smaller by about $\sqrt{2}$, which noise images of this dimension almost never have. Spherical interpolation moves along the great circle between the two codes instead,

$$\mathbf{z}_u = \frac{\sin\big((1-u)\omega\big)\,\mathbf{z}_0 + \sin(u\omega)\,\mathbf{z}_1}{\sin\omega}, \qquad \cos\omega = \frac{\mathbf{z}_0^\top \mathbf{z}_1}{\|\mathbf{z}_0\|\|\mathbf{z}_1\|}, \tag{17.39}$$

which keeps the norm close to that of the endpoints. We decode eight points between the 50-step codes of the first two test images.

```
z0, z1 = codes[50][0], codes[50][1]
omega = torch.arccos(((z0 * z1).sum() / (z0.norm() * z1.norm())).clamp(-1, 1))
u = torch.linspace(0, 1, 8, device=z0.device).reshape(-1, 1, 1, 1)
zs = (torch.sin((1 - u) * omega) * z0 + torch.sin(u * omega) * z1
      ) / torch.sin(omega)
d2l.show_gray(d2l.ddim_sample(pred, zs, lin, 50), 1, 8);
```



The row moves from a reconstruction of the first test image to one of the second, and every intermediate code decodes to a plausible item.

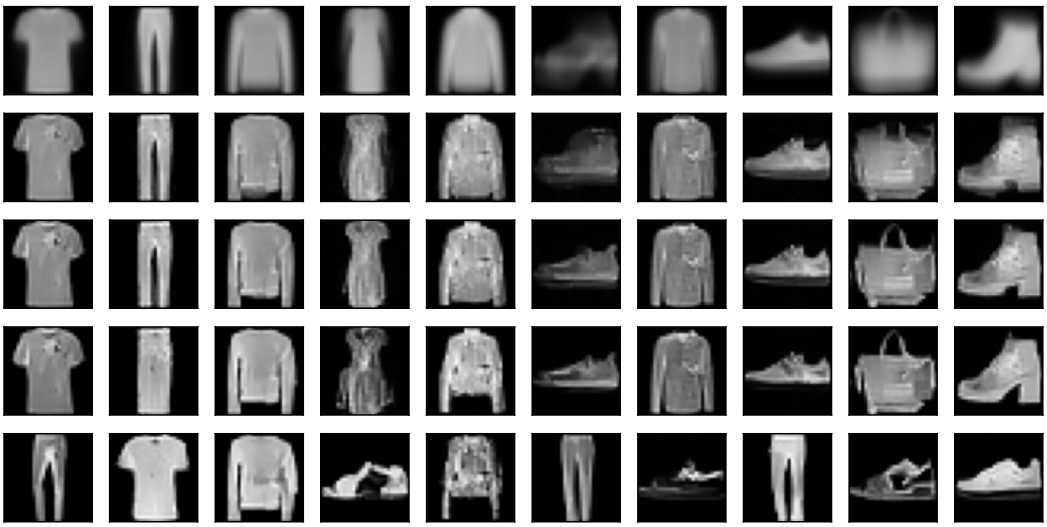
17.4.3.2 Editing by Partial Noising

Inversion runs an image all the way to noise; stopping part of the way gives an editor. C. Meng, He, et al. (2022) noise a rough input, such as a sketch or a crude collage, to an intermediate time t_0 with $\mathbf{x}_{t_0} = \alpha_{t_0}\mathbf{y} + \sigma_{t_0}\epsilon$, and run the sampler from t_0 back to the data. The noise hides the details of the input that real images do not have, the sampler replaces them by details that real images do have, and the coarse layout survives. The time t_0 sets the balance between fidelity to the input and realism.

Our rough inputs are the ten class-mean images, blurred silhouettes that no real garment looks like, each noised a hundred times to $t_0 \in \{0.3, 0.5, 0.7, 1\}$ and integrated back with 25 DDIM steps. The cell prints the squared distance per pixel to the input (fidelity) and the FD of the 1,000 outputs (realism); the row $t_0 = 0$ holds the inputs.

```
means = torch.stack([X[y == c].mean(0) for c in range(10)])
inputs = means.repeat(100, 1, 1, 1)                 # 1,000 inputs
torch.manual_seed(2)
eps = torch.randn_like(inputs)
edits = {0.0: inputs}
for t0 in (0.3, 0.5, 0.7, 1.0):
    x_t0 = lin.sample(inputs, torch.full((1000,), t0, device=X.device), eps)
    edits[t0] = d2l.ddim_sample(pred, x_t0, lin, 25, t_start=t0).clamp(-1, 1)
for t0, x in edits.items():
    print(f't0 = {t0:.1f}: squared distance to the input '
          f'{((x - inputs) ** 2).mean():.3f}, '
          f'FD {d2l.fid_score(features(x), f_test):.2f}')
d2l.show_gray(torch.cat([x[:10] for x in edits.values()]), 5, 10);
```

```
t0 = 0.0: squared distance to the input 0.000, FD 86.76
t0 = 0.3: squared distance to the input 0.019, FD 31.56
t0 = 0.5: squared distance to the input 0.046, FD 12.99
t0 = 0.7: squared distance to the input 0.090, FD 5.23
t0 = 1.0: squared distance to the input 0.435, FD 3.15
```



The distance to the input grows with t_0, and the FD falls from far above that of any sampler toward that of fresh samples. At $t_0 = 0.3$ each output keeps the silhouette of its mean image and gains the seams, prints and edges of a real garment; at 0.5 and 0.7 class and pose survive while the details vary; at $t_0 = 1$ the output is an unrelated sample.

17.4.3.3 Inpainting and Posterior Sampling

Suppose that we know part of an image, the upper half, say, and want the model to fill in the rest. Write $\mathbf{m}$ for the mask that is one on the known pixels and $\mathbf{y}$ for their values. At every level we know what the known pixels of $\mathbf{x}_s$ should look like, $\alpha_s\mathbf{y}$ plus noise of scale σ_s, so replacement inpainting (Y. Song, Sohl-Dickstein, et al. 2021; Lugmayr et al. 2022) takes each sampler step as usual and then overwrites the known pixels with a fresh noisy copy of $\mathbf{y}$ at the new level,

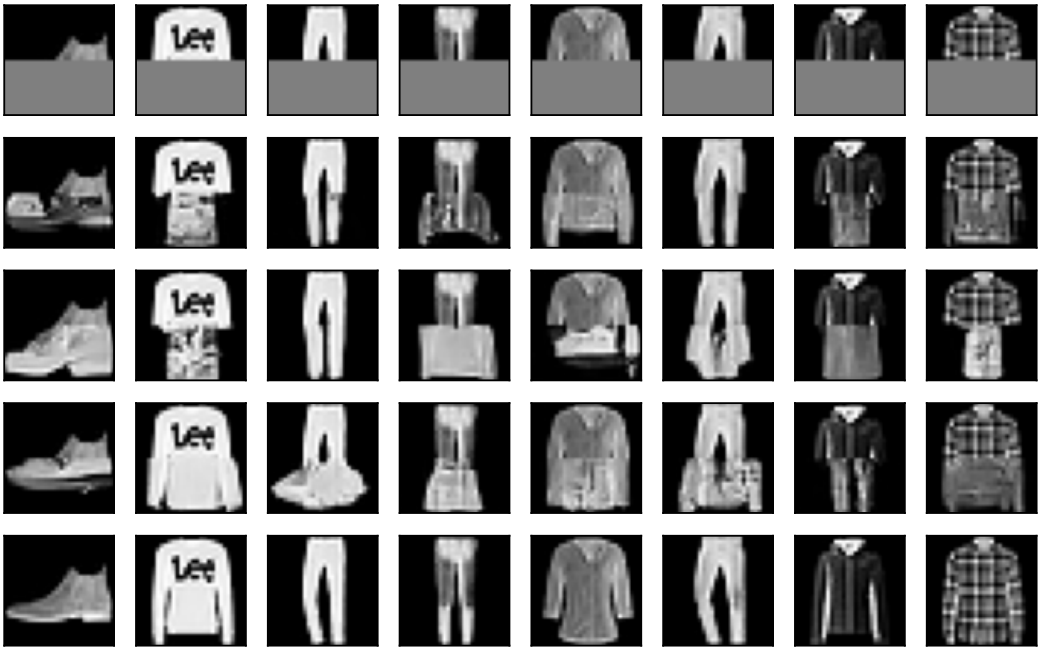
$$\mathbf{x}_s \leftarrow \mathbf{m} \odot \left(\alpha_s\mathbf{y} + \sigma_s\xi\right) + (\mathbf{1} - \mathbf{m}) \odot \mathbf{x}_s. \tag{17.40}$$

`inpaint` therefore runs the sampler one step at a time.

```
@torch.no_grad()
def inpaint(pred, known, mask, path, steps):
    """DDIM from noise that resets the known pixels (mask 1) at each level."""
    a, s = path.grid(steps)
    x = torch.randn_like(known)
    for i in range(steps):
        x0, eps = pred(x, a[i], s[i])
        x = a[i + 1] * x0 + s[i + 1] * eps            # one DDIM step, eta = 0
        noised = a[i + 1] * known + s[i + 1] * torch.randn_like(x)
        x = mask * noised + (1 - mask) * x            # keep the known pixels
    return x
```

We keep the upper half of eight test images and ask for three completions of each, from different noise. The rows show the known halves, the three completions and the originals.

```
mask = torch.zeros(1, 1, 28, 28, device=X.device)
mask[..., :14, :] = 1                                 # the upper half is known
torch.manual_seed(3)
fills = inpaint(pred, X_test[:8].repeat(3, 1, 1, 1), mask, lin, 50)
d2l.show_gray(torch.cat([X_test[:8] * mask, fills, X_test[:8]]), 5, 8);
```



Most completions continue the known half into a plausible garment of the same kind, some are malformed, and the three completions of one image differ. The boundary is not always clean: the network sees the known pixels only through noisy copies, and the halves can disagree along the seam (exercise 9 measures and repairs this).

Most measurements, such as a blurred or low-resolution copy, are not a subset of the pixels; for these, Bayes' rule adds the gradient of the measurement's log-likelihood to the score, as guidance did for a class label in Section 17.3, and Chung et al. (2023) evaluate that likelihood at the network's estimate $\hat{\mathbf{x}}_0(\mathbf{x}_t)$.

17.4.4 Summary

Training fixes only the distribution of the noisy point at each time, so one network serves a family of samplers, from DDIM, which is Euler's method for the probability-flow ODE in the coordinates $(\mathbf{x}/\alpha, \sigma/\alpha)$, to a stochastic sampler with the posterior variance. Fresh noise corrects errors but needs short steps, so the deterministic sampler wins at small budgets: on Fashion-MNIST fifty DDIM steps came within about 1.5 times the FD of the thousand-step chain, and Heun's method matched the chain at about fifty evaluations, with the grid mattering as much as the method. Run backwards, the deterministic sampler gives every image a code, and with it interpolation, editing and inpainting.

We have trained every model so far by an unweighted squared error on a target we picked, without asking why this loss is the right one or how the choices of target, path and weighting compare. Section 17.5 derives the loss from the likelihood and measures the choices.

17.4.5 Exercises

1. **Admissible noise scales.** The family Equation 17.33 needs $\tilde{\sigma} \leq \sigma_s$.

1. Show that a strength η is admissible on a step from t to s exactly when $\eta^2 \le 1/(1-\kappa^2)$, so that $\eta = 1$ is admissible on every grid. At the largest admissible η, what does the step keep of $\mathbf{x}_t$ beyond $\hat{\mathbf{x}}_0$?
2. Show that the last step of `ddim_sample`, to $t = 0$, returns $\hat{\mathbf{x}}_0$ for every η.
3. Evaluate the fraction f of fresh noise at $\eta = 1$ on every step of the variance-preserving grids with 10 and with 200 steps. What does the $\eta = 1$ sampler carry from one level to the next on the ten-step grid?

2. **The forward step and the Markov member.** ▣ Given $\mathbf{x}_0$, draw $\mathbf{x}_t = \alpha_t \mathbf{x}_0 + \sigma_t \epsilon$ and $\mathbf{x}_s$ from Equation 17.33 with the same ϵ.
   1. Compute the mean and the covariance of $\mathbf{x}_t$ given $\mathbf{x}_s$ and $\mathbf{x}_0$ by conditioning the joint Gaussian.
   2. Show that the mean depends on $\mathbf{x}_0$ unless $\tilde{\sigma}$ takes its value at $\eta = 1$, and that the conditional is then the forward step $\mathbf{x}_t = (\alpha_t/\alpha_s)\mathbf{x}_s + \sqrt{\sigma_t^2 - (\alpha_t/\alpha_s)^2 \sigma_s^2}\,\nu$ of the noising chain of the previous section. Conclude that this chain is the only member of the family whose forward process is Markov.
   3. Show that with $\tilde{\sigma} = 0$ at every step the forward process is $\mathbf{x}_t = \alpha_t \mathbf{x}_0 + \sigma_t \epsilon$ with one ϵ shared by all times. Why can the regression not tell these forward processes apart?
3. **Gaussian data.** Let the data be one-dimensional with distribution $\mathcal{N}(0, v)$, and work in the coordinates $(\bar{x}, \bar{\sigma})$.
   1. Show that the exact noise estimate is $\hat{\epsilon} = \bar{\sigma}\bar{x}/(v + \bar{\sigma}^2)$, that one deterministic stride from $\bar{\sigma}$ to $\bar{\sigma}'$ multiplies $\bar{x}$ by $(v + \bar{\sigma}\bar{\sigma}')/(v + \bar{\sigma}^2)$, and that the exact flow multiplies it by $\sqrt{(v + \bar{\sigma}'^2)/(v + \bar{\sigma}^2)}$.
   2. Show that the factor of the stride never exceeds the exact one, with equality only for $\bar{\sigma}' = \bar{\sigma}$. What does this imply for the variance of deterministic samples of Gaussian data on any grid?
   3. Take $v = 0.25$, the variance of one coordinate of a mode of `mix`. Compute the standard deviation of deterministic samples from $x_1 \sim \mathcal{N}(0, 1)$ on the variance-preserving grids with 200, 50, 20, 10 and 5 steps, and compare it with 0.5 and with the spreads printed in this section.
4. code **The strength η and the code.** Use the image model of this section.
   1. Compute the FD of 1,000 samples of `ddim_sample` for $\eta \in \{0, 0.25, 0.5, 1\}$ at 10, 20 and 50 steps.
   2. Fix one noise image $\mathbf{x}_1$ and draw 16 samples from it for each η with different random seeds. Compute the standard deviation of each pixel across the 16 samples and average it over the pixels. How much does the image depend on $\mathbf{x}_1$ at each η?
   3. Which η would you use at each budget, and which if you need the sample to be a function of its code?
5. code **Inversion on both examples.** Encode 20,000 draws of `mix` with `ddim_invert` and `pred_exact` on the variance-preserving grid with 10, 20, 50, 100 and 200 steps, and decode each set of codes with `ddim_sample` on the same grid.
   1. How does the mean reconstruction error depend on the number of steps? Does it fall in proportion to the step size?
   2. Compute the mean and the covariance of the 200-step codes. Are they those of standard Gaussian noise?
   3. Repeat part (a) for the 100 test images with `pred` on the linear path, adding 200 steps, and compare the trend with Table 2 of J. Song et al. (2021) .
6. [extended] **A second-order sampler in log-SNR.** ▣▣ DPM-Solver-2 (Lu, Zhou, et al. 2022) takes a step from $\bar{\sigma}$ to $\bar{\sigma}'$ in two evaluations, the second at the midpoint $\bar{\sigma}_m =$

$\sqrt{\bar\sigma\bar\sigma'}$ in $\log\bar\sigma$: $\bar{\mathbf{x}}_m = \bar{\mathbf{x}} + (\bar\sigma_m - \bar\sigma)\,\hat\epsilon(\bar{\mathbf{x}}, \bar\sigma)$ and $\bar{\mathbf{x}}' = \bar{\mathbf{x}} + (\bar\sigma' - \bar\sigma)\,\hat\epsilon(\bar{\mathbf{x}}_m, \bar\sigma_m)$.

1. Let $h = \log(\bar\sigma/\bar\sigma')$. Write the exact change of $\bar{\mathbf{x}}$ over the step as an integral of $\hat\epsilon$ along the trajectory, and show that the local error of the update is $O(\bar\sigma h^3)$ while that of the Euler step is $O(\bar\sigma h^2)$. Assume that $\hat\epsilon$ is a smooth function of $\log\bar\sigma$ along the trajectory and Lipschitz in $\bar{\mathbf{x}}$.
2. Implement the update with the interface `pred(x, alpha, sigma)`, using $\alpha = 1/\sqrt{1+\bar\sigma^2}$ and $\sigma = \bar\sigma\alpha$, and take the last step to $\bar\sigma = 0$ with Euler's method. Compare it with `ode_sample` and `heun_sample` at about 11, 21 and 51 evaluations, on the mixture with `pred_exact` by the distance to a 1,000-step reference, and on images by the FD.
3. Repeat with levels evenly spaced in $\log\bar\sigma$ between 80 and 0.002. Which combination of method and grid is best at each budget?

7. code **Fewer ancestral steps.** Run `ancestral_sample` and `ddim_sample` with $\eta = 1$ on the mixture with `pred_exact`, on the variance-preserving grids with 200, 50, 20 and 10 steps, from 20,000 starting points.
   1. Compare the `mix.summary` statistics of the two samplers with those of the data. Where does each sampler break first: in the mode weights, in the spread or in the stray fraction?
   2. Relate the change of the spread to the variance error of one ancestral step (exercise 4) and to the factor σ_s^2/σ_t^2 between the two samplers' variances.
   3. Show 16 images of each sampler at 50 steps on the image model, and explain the speckle of the ancestral ones.
8. **The optimal reverse variance.** ⊠ A stochastic step from t to s moves to the posterior mean $\alpha_s\hat{\mathbf{x}}_0 + \sigma_s\kappa\hat\epsilon$ and adds Gaussian noise of variance v per coordinate. Ho et al. (2020) tried the forward-step variance and the posterior variance; Bao et al. (2022) found the best v for any data. Assume exact estimates, and let $\bar e_t = \mathbb{E}\|\epsilon - \hat\epsilon\|^2/d$ be the error per coordinate of the exact noise estimate at time t in d dimensions.
   1. Exercise 5a shows that the true reverse transition $q(\mathbf{x}_s \mid \mathbf{x}_t)$ has covariance $\tau^2\mathbf{I} + c^2\mathrm{Cov}(\mathbf{x}_0 \mid \mathbf{x}_t)$, where c is the coefficient of $\mathbf{x}_0$ in the mean of $q(\mathbf{x}_s \mid \mathbf{x}_t, \mathbf{x}_0)$. Write that mean as $\alpha_s\mathbf{x}_0 + \sigma_s\kappa\epsilon$ with $\epsilon = (\mathbf{x}_t - \alpha_t\mathbf{x}_0)/\sigma_t$, and conclude that $\tau^2 = \tilde\sigma^2$, the posterior variance, and $c = k = \alpha_s - \sigma_s\kappa\,\alpha_t/\sigma_t$. What is the mean of the true reverse transition?
   2. Show that the v that minimizes the expected KL divergence from $q(\mathbf{x}_s \mid \mathbf{x}_t)$ to the Gaussian step is $v^\star = \tilde\sigma^2 + k^2\bar\sigma_t^2\bar e_t$, and that $\bar e_t = 1 - \sigma_t^2\,\mathbb{E}\|\nabla\log p_t(\mathbf{x}_t)\|^2/d$, so that $v^\star$ follows from the score alone.
   3. Show that $v^\star$ is the posterior variance when the data are one point, and the forward-step variance when the data are standard Gaussian on the variance-preserving path.
   4. (Optional, code.) Estimate $\bar e_t$ on the mixture with `pred_exact` on the 50-step variance-preserving grid, and compare the spread of 20,000 samples of the stochastic step with the three variances.
9. code **Inpainting with harmonization.** ⊠ The completions of `inpaint` can disagree with the known half along the boundary.
   1. Measure the seam: the mean absolute difference between rows 13 and 14 of the completions, against the same quantity for real test images.
   2. RePaint (Lugmayr et al. 2022) lets the two halves exchange information: after each step from t to s, it noises the result back to t with the forward step and repeats the step, r times per level. Implement it with $r \in \{1, 5, 10\}$ at 50 levels and measure the seam again. What does each repetition cost?
   3. Draw five completions of one test image for each r. How varied are they?

10. code **A rescaled schedule.** ⊠ Let $c_t = 1+t(1-t)$ and define a path with $\alpha'_t = c_t(1-t)$ and $\sigma'_t = c_t t$, which has the same endpoints as the linear path.
    1. Show that the two paths have the same log-SNR at every t, and that the deterministic DDIM steps of Equation 17.34 on the two paths produce the same $\hat{\mathbf{x}}_0$ at every step and states that differ by the factor c_t. Check this with `ddim_sample` and `pred` on the image model.
    2. Implement Euler's method on the velocity, $\mathbf{x} \leftarrow \mathbf{x} + (s-t)\,(\dot\alpha_t \hat{\mathbf{x}}_0 + \dot\sigma_t \hat\epsilon)$, with the rates from `path.rates`, on both paths with 10 steps. Show that the samples differ, and explain why with exercise 7.
    3. Gao et al. (2025) conclude that DDIM is unaffected by such a rescaling. Which of the two choices, the path or the grid of times, can change the samples of DDIM?
11. code **Editing toward another class.** Train the class-conditional U-Net of Section 17.3 and steer with `d2l.make_pred(d2l.cfg(ema, c, gamma), lin, 'velocity')`, where `c` holds one label per image.
    1. Noise 100 test images of trousers to $t_0 \in \{0.3, 0.5, 0.7, 0.9\}$ and integrate back with 25 DDIM steps toward the label of dresses, at $\gamma = 1$ and $\gamma = 2$. Report the fraction of outputs that `cnn` classifies as dresses and the squared distance per pixel to the input, as in the editing cell.
    2. From pure noise, sample with the label of trousers and $\gamma = 1$ from $t = 1$ down to t^*, then with the label of dresses and the two scales of (a) down to $t = 0$ (`ddim_sample` takes `t_start` and `t_end`), for $t^* \in \{0.3, 0.5, 0.7, 0.9\}$. Plot the fraction classified as dresses against t^*, next to the curve of (a) against t_0, and compare both with the commitment time of Exercise 6 of the mixture.
    3. Show that with an exact network the two experiments at the same γ and $t_0 = t^*$ produce outputs with the same distribution. Which differences between the curves remain, and where do they come from?

    Part (a) is adapted from Berkeley CS180/280A, Project 5A (Fall 2025), Section 1.7.3.

17.5 Likelihood, Weighting and the Design Space

For four sections we have trained networks by regression onto a per-example target and accepted the result because it worked: samples of the mixture land on its three modes with the right weights, and the garments of Section 17.3 look like garments. We have never asked what probability the model assigns to a data point, a density model's first question and one the adversarial models of Chapter 16 could not answer. Is "predict the noise" maximum likelihood in disguise, and what negative log-likelihood does it give the mixture, whose true value is about 2.5 nats per point? And which of the choices made along the way (the target, the weight of each noise level, the schedule) matter?

Sohl-Dickstein et al. (2015) and Ho et al. (2020) derived the model as a latent-variable model whose latents are the noisy copies of a data point, with a lower bound on the log-likelihood that is a sum of noise regressions, one per noise level, each with its own weight; the loss of Section 17.2 sets every weight to one. Kingma and Gao (2023) showed that every diffusion objective, flow matching included, is the same integral over the log signal-to-noise ratio with its own weighting. The section ends with the design choices of Karras et al. (2022), noise levels as image scales, and four recipes on Fashion-MNIST.

```
torch.set_float32_matmul_precision('high')  # tf32 matmuls on tensor cores
```

17.5.1 The Likelihood Bound

Read backward, the chain of Section 17.2 is a generative model with T layers of latent variables, whose likelihood we can bound.

17.5.1.1 The Chain as a Latent-Variable Model

Step k of the forward chain shrinks the current point and adds noise, $\mathbf{x}_k = \sqrt{1-\beta_k}\,\mathbf{x}_{k-1} + \sqrt{\beta_k}\,\epsilon_k$, and the steps compose into one Gaussian draw, $\mathbf{x}_k = \alpha_k \mathbf{x}_0 + \sigma_k \epsilon$ with $\alpha_k^2 = \prod_{j\le k}(1-\beta_j)$ and $\sigma_k^2 = 1-\alpha_k^2$: the VP path at the times $t_k = k/T$. We write q for the distributions of this forward chain. The generative model runs it backward through learned Gaussian steps,

$$p_\theta(\mathbf{x}_{0:T}) = p(\mathbf{x}_T)\prod_{k=1}^{T} p_\theta(\mathbf{x}_{k-1} \mid \mathbf{x}_k), \qquad p_\theta(\mathbf{x}_{k-1} \mid \mathbf{x}_k) = \mathcal{N}\big(\mu_\theta(\mathbf{x}_k, k),\, \nu_k \mathbf{I}\big), \tag{17.41}$$

with $p(\mathbf{x}_T) = \mathcal{N}(\mathbf{0}, \mathbf{I})$, a mean μ_θ computed by a network, and fixed variances ν_k (σ_k remains the path's noise scale). The density of $\mathbf{x}_0$ integrates this joint over $\mathbf{x}_{1:T}$, which we cannot compute, but any distribution over the latents gives a lower bound on it, the evidence lower bound (ELBO) of Section 36.7. The chain supplies one, the forward process itself. As an upper bound on the negative log-likelihood,

$$-\log p_\theta(\mathbf{x}_0) \le \mathbb{E}_{q(\mathbf{x}_{1:T}\mid\mathbf{x}_0)}\left[-\log \frac{p_\theta(\mathbf{x}_{0:T})}{q(\mathbf{x}_{1:T} \mid \mathbf{x}_0)}\right] =: L(\mathbf{x}_0). \tag{17.42}$$

The gap is $D_{\mathrm{KL}}\big(q(\mathbf{x}_{1:T} \mid \mathbf{x}_0) \,\|\, p_\theta(\mathbf{x}_{1:T} \mid \mathbf{x}_0)\big)$, so the bound is tight when the learned reverse chain, given $\mathbf{x}_0$, retraces the forward chain.

Substituting the factorizations of both chains splits the bound into one term per step. With $\mathbb{E}_q$ the expectation over $q(\mathbf{x}_{1:T} \mid \mathbf{x}_0)$,

$$\begin{aligned} L(\mathbf{x}_0) &= \mathbb{E}_q\Big[-\log p(\mathbf{x}_T) - \sum_{k=1}^{T} \log \frac{p_\theta(\mathbf{x}_{k-1} \mid \mathbf{x}_k)}{q(\mathbf{x}_k \mid \mathbf{x}_{k-1})}\Big] \\ &= \mathbb{E}_q\Big[-\log p(\mathbf{x}_T) - \sum_{k=2}^{T} \log \Big(\frac{p_\theta(\mathbf{x}_{k-1} \mid \mathbf{x}_k)}{q(\mathbf{x}_{k-1} \mid \mathbf{x}_k, \mathbf{x}_0)} \cdot \frac{q(\mathbf{x}_{k-1} \mid \mathbf{x}_0)}{q(\mathbf{x}_k \mid \mathbf{x}_0)}\Big) - \log \frac{p_\theta(\mathbf{x}_0 \mid \mathbf{x}_1)}{q(\mathbf{x}_1 \mid \mathbf{x}_0)}\Big] \\ &= \mathbb{E}_q\Big[\log \frac{q(\mathbf{x}_T \mid \mathbf{x}_0)}{p(\mathbf{x}_T)} + \sum_{k=2}^{T} \log \frac{q(\mathbf{x}_{k-1} \mid \mathbf{x}_k, \mathbf{x}_0)}{p_\theta(\mathbf{x}_{k-1} \mid \mathbf{x}_k)} - \log p_\theta(\mathbf{x}_0 \mid \mathbf{x}_1)\Big]. \end{aligned} \tag{17.43}$$

The second line reverses each forward step by Bayes' rule with $\mathbf{x}_0$ held fixed: by the Markov property $q(\mathbf{x}_k \mid \mathbf{x}_{k-1}) = q(\mathbf{x}_k \mid \mathbf{x}_{k-1}, \mathbf{x}_0)$, which equals $q(\mathbf{x}_{k-1} \mid \mathbf{x}_k, \mathbf{x}_0)\, q(\mathbf{x}_k \mid \mathbf{x}_0)/q(\mathbf{x}_{k-1} \mid \mathbf{x}_0)$. In the third line the ratios of marginals telescope to $q(\mathbf{x}_1 \mid \mathbf{x}_0)/q(\mathbf{x}_T \mid \mathbf{x}_0)$, whose numerator cancels the $k=1$ denominator. Each remaining logarithm is, in expectation, a divergence:

$$L(\mathbf{x}_0) = \underbrace{D_{\mathrm{KL}}(q(\mathbf{x}_T \mid \mathbf{x}_0) \,\|\, p(\mathbf{x}_T))}_{L_T} + \sum_{k=2}^{T} \underbrace{\mathbb{E}_q \, D_{\mathrm{KL}}(q(\mathbf{x}_{k-1} \mid \mathbf{x}_k, \mathbf{x}_0) \,\|\, p_\theta(\mathbf{x}_{k-1} \mid \mathbf{x}_k))}_{L_{k-1}} + \underbrace{\mathbb{E}_q[-\log p_\theta(\mathbf{x}_0 \mid \mathbf{x}_1)}_{L_0} \tag{17.44}$$

The middle terms compare each learned step with the *forward posterior* $q(\mathbf{x}_{k-1} \mid \mathbf{x}_k, \mathbf{x}_0)$, the reversal of step k for a known clean point. Conditioning on $\mathbf{x}_0$ makes it computable: as a product of two Gaussians in $\mathbf{x}_{k-1}$, the forward step and $q(\mathbf{x}_{k-1} \mid \mathbf{x}_0)$, it is Gaussian (8 completes the square),

$$q(\mathbf{x}_{k-1} \mid \mathbf{x}_k, \mathbf{x}_0) = \mathcal{N}(\tilde{\mu}_k, \tilde{\beta}_k \mathbf{I}), \qquad \tilde{\mu}_k = \frac{\alpha_{k-1}\beta_k}{\sigma_k^2}\,\mathbf{x}_0 + \frac{\sqrt{1-\beta_k}\,\sigma_{k-1}^2}{\sigma_k^2}\,\mathbf{x}_k, \qquad \tilde{\beta}_k = \frac{\beta_k\,\sigma_{k-1}^2}{\sigma_k^2}. \tag{17.45}$$

The mean is a weighted average of where the clean point and the noisy point put $\mathbf{x}_{k-1}$, and the variance is below the forward step's β_k: knowing both ends pins the middle down.

The smallest case shows the three kinds of terms. With one coordinate, $T = 2$ and $\beta_1 = \beta_2 = 1/2$, the *prior term* L_2 compares the chain's end $\mathcal{N}(\alpha_2 x_0, \sigma_2^2)$ with $\mathcal{N}(0, 1)$ and by the Gaussian formula Equation 16.8 equals $x_0^2/8 + 0.019$: two large steps leave much of x_0 behind. The *denoising term* L_1 compares the learned step from x_2 to x_1 with the forward posterior, and the *reconstruction term* L_0 scores the last step. At $T = 1000$ the prior term is nearly zero and the denoising terms carry the training signal.

Averaged over the data, the bound is at least the cross-entropy $\mathbb{E}_p[-\log p_\theta(\mathbf{x}_0)] = H(p) + D_{\mathrm{KL}}(p \,\|\, p_\theta)$: a model that drops a mode pays on every point of it.

17.5.1.2 From the Bound to Weighted Regression

Each denoising term compares two Gaussians whose variances do not depend on the network, so by Equation 16.8 only the distance between the means involves θ:

$$L_{k-1} = \mathbb{E}_q\Big[\frac{1}{2\nu_k}\big\|\tilde{\mu}_k(\mathbf{x}_k, \mathbf{x}_0) - \mu_\theta(\mathbf{x}_k, k)\big\|^2\Big] + \text{const}, \tag{17.46}$$

where the constant vanishes for $\nu_k = \tilde{\beta}_k$: each step asks the network for the mean of the forward posterior.

The network does not have to output that mean directly. Substituting $\mathbf{x}_0 = (\mathbf{x}_k - \sigma_k \epsilon)/\alpha_k$ into Equation 17.45 and using $\sigma_k^2 = \beta_k + (1-\beta_k)\sigma_{k-1}^2$, the coefficients of $\mathbf{x}_k$ combine into $1/\sqrt{1-\beta_k}$ and the mean becomes

$$\tilde{\mu}_k = \frac{1}{\sqrt{1-\beta_k}}\Big(\mathbf{x}_k - \frac{\beta_k}{\sigma_k}\,\epsilon\Big). \tag{17.47}$$

Everything here but ϵ is known to a network that receives $\mathbf{x}_k$ and k, so we let it predict the noise and form its mean the same way,

$$\mu_\theta(\mathbf{x}_k, k) = \frac{1}{\sqrt{1-\beta_k}}\Big(\mathbf{x}_k - \frac{\beta_k}{\sigma_k}\,\hat{\epsilon}_\theta(\mathbf{x}_k, k)\Big), \tag{17.48}$$

and the difference of the two means is a multiple of the noise error:

$$L_{k-1} = \mathbb{E}_{\mathbf{x}_0,\epsilon}\Big[\frac{\beta_k^2}{2\nu_k(1-\beta_k)\sigma_k^2}\,\big\|\epsilon - \hat{\epsilon}_\theta(\alpha_k\mathbf{x}_0 + \sigma_k\epsilon, k)\big\|^2\Big] + \text{const.} \tag{17.49}$$

Each term of the bound is the noise regression of Section 17.2 at one level, with a weight, for $\nu_k = \tilde{\beta}_k$ equal to $\beta_k/(2(1-\beta_k)\sigma_{k-1}^2)$. Over step k the signal-to-noise ratio α_k^2/σ_k^2 falls by the factor $\sigma_k^2/((1-\beta_k)\sigma_{k-1}^2) = 1 + \beta_k/((1-\beta_k)\sigma_{k-1}^2)$, so with $\lambda_k = \log(\alpha_k^2/\sigma_k^2)$ the weight is

$$\frac{\beta_k}{2(1-\beta_k)\sigma_{k-1}^2} = \tfrac{1}{2}(e^{\lambda_{k-1}-\lambda_k} - 1) \approx \tfrac{1}{2}(\lambda_{k-1} - \lambda_k).$$

The bound charges every step by how much it lowers the log-SNR. The cell prints these weights along our VP chain ($T = 1000$).

```
vp = d2l.GaussianPath.vp()
a, s = vp.grid(1000)
a, s = [float(v) for v in a[::-1]], [float(v) for v in s[::-1]]  # k = 0..T
for k in (2, 10, 100, 500, 1000):
    beta = 1 - (a[k] / a[k - 1]) ** 2
    beta_tilde = beta * s[k - 1] ** 2 / s[k] ** 2
    weight = lambda nu: beta ** 2 / (2 * nu * (1 - beta) * s[k] ** 2)
    dlam = math.log((a[k - 1] * s[k]) ** 2 / (a[k] * s[k - 1]) ** 2)
    print(f'k = {k:4d}: weight {weight(beta):.4f} (nu = beta), '
          f'{weight(beta_tilde):.4f} (nu = beta_tilde), '
          f'half log-SNR step {dlam / 2:.4f}')
```

```
k =    2: weight 0.2708 (nu = beta), 0.5906 (nu = beta_tilde), half log-SNR step 0.3899
k =   10: weight 0.0725 (nu = beta), 0.0848 (nu = beta_tilde), half log-SNR step 0.0783
k =  100: weight 0.0100 (nu = beta), 0.0102 (nu = beta_tilde), half log-SNR step 0.0101
k =  500: weight 0.0055 (nu = beta), 0.0055 (nu = beta_tilde), half log-SNR step 0.0055
k = 1000: weight 0.0101 (nu = beta), 0.0101 (nu = beta_tilde), half log-SNR step 0.0100
```

Where the steps are small, the weight for $\nu_k = \tilde{\beta}_k$ equals half the log-SNR step; for $\nu_k = \beta_k$ it differs only at the start. Both are largest at the first steps, where the log-SNR falls by $\log(k/(k-1))$ per step, and stay near 0.01 or below after the first few dozen. The simple loss of Section 17.2,

$$L_{\text{simple}}(\theta) = \mathbb{E}_{k,\mathbf{x}_0,\epsilon}\,\big\|\epsilon - \hat{\epsilon}_\theta(\alpha_k\mathbf{x}_0 + \sigma_k\epsilon, k)\big\|^2,$$

with k uniform on $\{1, \dots, T\}$, sets every weight to one and so neglects the least noisy steps relative to the bound. Ho et al. (2020) found that it nevertheless gives better samples. Those steps hold the finest details of an image (Section 17.5.3.2), which a likelihood counts in full and a viewer hardly notices.

As the steps shrink, the weighted sum becomes an integral over the log-SNR,

$$L(\mathbf{x}_0) \to \frac{1}{2}\int_{\lambda_{\min}}^{\lambda_{\max}} \mathbb{E}_\epsilon\left\|\epsilon - \hat{\epsilon}_\theta(\mathbf{x}_\lambda, \lambda)\right\|^2 d\lambda + L_T + L_0, \tag{17.50}$$

where $\mathbf{x}_\lambda$ is the point of the path with log-SNR λ (Kingma et al. 2021), from $\lambda_{\max}$ at the smallest noise level to $\lambda_{\min}$ at the path's end. The limit needs steps small in λ, which the chain's grid lacks at the data end, where the log-SNR falls by $\log 2$ from the first step to the second whatever T is; there the chain's bound is loose (2).

17.5.1.3 A Number for the Mixture

Two numbers measure the likelihood of a trained network: the bound Equation 17.50, which belongs to the stochastic reverse process, and the density of the deterministic probability-flow ODE of Section 17.2, by the change of variables; for the exact field they coincide. We train a noise predictor on the VP path with the mixture recipe of Section 17.2 for $20{,}000$ steps.

```
device = d2l.try_gpu()
mix = d2l.GaussianMixture.three_modes(device)

torch.manual_seed(0)
_, ema, _ = d2l.train_regression(
    d2l.TimeMLP(256, 3).to(device), d2l.regression_loss(vp, 'eps'),
    lambda: mix.sample(512), 20_000, cosine=True, ema_half_life=700)
```

Along a trajectory of $d\mathbf{x}/dt = \mathbf{u}_t(\mathbf{x})$ the log-density changes at the rate $-\nabla \cdot \mathbf{u}_t$, the instantaneous change of variables of Section 38.2.3. Integrating from a data point at a small time $t_{\min}$ to $t = 1$, where the density is a standard Gaussian, gives

$$\log p(\mathbf{x}_{t_{\min}}) = \log \mathcal{N}(\mathbf{x}_1; \mathbf{0}, \mathbf{I}) + \int_{t_{\min}}^{1} \nabla \cdot \mathbf{u}_t(\mathbf{x}_t)\, dt.$$

`pf_nll` integrates the point and the divergence by fourth-order Runge–Kutta on 100 steps that crowd toward the data; the velocity comes from the dictionary of Section 17.2, the divergence is the trace of the 2×2 Jacobian, one forward-mode derivative per coordinate, and the reference is the mixture's exact noise prediction. We start at $t_{\min} = 0.001$, where the noise is about 1% of the signal.

```
def eps_exact(x, t):
    """E[eps | x_t] of the mixture, from its posterior mean."""
    x0 = mix.posterior_mean(x, vp.alpha(t), vp.sigma(t))
    return vp.x0_eps('x0', x0, x, t)[1]

def pf_nll(eps_fn, path, x, t_min=1e-3, steps=100):
    """-log p(x) under the probability-flow ODE, RK4 from t_min to 1."""
    def f(x, t):                              # velocity and its divergence
        t = torch.full((len(x),), t, device=x.device)
        def u(x):                             # the velocity, by the dictionary
```

```
            x0, eps = path.x0_eps('eps', eps_fn(x, t), x, t)
            return path.target('velocity', x0, eps, t)
        e = torch.eye(x.shape[1], device=x.device)
        jvps = [torch.func.jvp(u, (x,), (e_i.expand_as(x),)) for e_i in e]
        return jvps[0][0], sum(j[1][:, i] for i, j in enumerate(jvps))
    ts = (t_min + (1 - t_min) * torch.linspace(0, 1, steps + 1) ** 2).tolist()
    div = torch.zeros(len(x), device=x.device)
    for ta, tb in zip(ts[:-1], ts[1:]):       # from the data to the noise
        h = tb - ta
        k1, d1 = f(x, ta)
        k2, d2 = f(x + h / 2 * k1, ta + h / 2)
        k3, d3 = f(x + h / 2 * k2, ta + h / 2)
        k4, d4 = f(x + h * k3, tb)
        x = x + h / 6 * (k1 + 2 * k2 + 2 * k3 + k4)
        div = div + h / 6 * (d1 + 2 * d2 + 2 * d3 + d4)
    return (x ** 2).sum(1) / 2 + x.shape[1] / 2 * math.log(2 * math.pi) - div
```

We evaluate both fields on 4,000 fresh draws.

```
torch.manual_seed(1)
test = mix.sample(20_000)
nll = -mix.log_prob(test)
mean_se = lambda v: f'{v.mean():.4f} +- {v.std() / math.sqrt(len(v)):.4f}'
with torch.no_grad():
    pf_exact, pf_net = (pf_nll(f, vp, test[:4000]) for f in (eps_exact, ema))
print(f'true NLL {mean_se(nll[:4000])} nats')
print(f'flow, exact field - true: {mean_se(pf_exact - nll[:4000])}')
print(f'flow, network - exact field: {mean_se(pf_net - pf_exact)}')
```

```
true NLL 2.4831 +- 0.0171 nats
flow, exact field - true: 0.0001 +- 0.0001
flow, network - exact field: 0.0025 +- 0.0013
```

The exact field's flow reproduces the true 2.5 nats per point to about four decimals, which checks the integrator; the trained network's comes within about 0.01 nats of it (a few thousandths in our runs). The network has learned the density, not only how to sample from it.

The prior term of the bound is about $\frac{1}{2}\alpha_1^2\|\mathbf{x}_0\|^2 \approx 2\cdot 10^{-4}$ nats. The last step outputs a Gaussian around $\hat{\mathbf{x}}_0$ with the variance $e^{-\lambda_{\max}} = \sigma^2/\alpha^2$ of the smallest noise level, and since $\mathbf{x}_0 - \hat{\mathbf{x}}_0 = (\sigma/\alpha)(\hat{\epsilon} - \epsilon)$,

$$L_0 = \frac{d}{2}\log\left(2\pi e^{-\lambda_{\max}}\right) + \frac{1}{2}\,\mathbb{E}\left\|\epsilon - \hat{\epsilon}_\theta(\mathbf{x}_{t_{\min}}, t_{\min})\right\|^2.$$

In two dimensions at a noise level of 0.01 this is about $-7.3+1 = -6.3$ nats, so the integral alone, about 9 nats, is not a likelihood; only the sum is.

bound estimates the integral by Monte Carlo, with λ uniform and stratified. Pinning the bound down to 0.01 nats takes a few million network evaluations, but the difference between two predictors on shared points, levels and noise takes far fewer (Section 36.5.2.6), so bound evaluates a list of predictors on shared draws.

```
@torch.no_grad()
def bound(eps_fns, path, x0, draws, t_min=1e-3):
    """Per-point bound on -log p(x0) for each eps(x, t), on shared draws."""
    n, d = x0.shape
    a1, s1 = (f(torch.tensor(1.0)).item() for f in (path.alpha, path.sigma))
    lam_max = path.log_snr(torch.tensor(t_min)).item()
    lam_min = 2 * math.log(a1 / s1)               # the end of the path
    scale = (lam_max - lam_min) / (2 * draws)
    prior = (((a1 * x0) ** 2).sum(1) / 2         # KL to N(0, I)
             + d / 2 * (s1 ** 2 - 1 - 2 * math.log(s1)))
    t = torch.full((n,), t_min, device=x0.device)
    eps = torch.randn_like(x0)
    x = path.sample(x0, t, eps)                  # the reconstruction term
    out = [prior + d / 2 * math.log(2 * math.pi / math.exp(lam_max))
           + ((f(x, t) - eps) ** 2).sum(1) / 2 for f in eps_fns]
    for _ in range(draws):                       # log-SNR uniform, stratified
        u = (torch.randperm(n, device=x0.device)
             + torch.rand(n, device=x0.device)) / n
        t = path.time_at(lam_max - u * (lam_max - lam_min))
        eps = torch.randn_like(x0)
        x = path.sample(x0, t, eps)
        for i, f in enumerate(eps_fns):
            out[i] += scale * ((f(x, t) - eps) ** 2).sum(1)
    return out

b_exact, b_net = bound([eps_exact, ema], vp, test, 64)
print(f'bound, exact field - true: {mean_se(b_exact - nll)}')
print(f'bound, network - exact field: {mean_se(b_net - b_exact)}')
```

```
bound, exact field - true: 0.0045 +- 0.0148
bound, network - exact field: 0.0110 +- 0.0008
```

With the exact field the bound matches the true negative log-likelihood within its Monte Carlo error of about 0.015 nats: in the continuous limit the reverse process with the exact score has no gap. The network's bound lies about a hundredth of a nat above the exact field's (a paired difference resolved to about a thousandth), so on the mixture regression onto the noise is maximum likelihood to within about a hundredth of a nat.

17.5.2 Every Objective Is a Weighting

The bound weights each unit of log-SNR equally, the simple loss each step of the chain, and flow matching regresses a velocity on another path. In a common variable these are one objective with three weightings.

17.5.2.1 The Log-SNR Form

Each of our losses draws a time t, forms $\mathbf{x}_t$, and takes the squared error of a target. Every target is affine in the noise prediction at fixed $\mathbf{x}_t$, so each squared error is $c(t)\,\|\hat{\epsilon} - \epsilon\|^2$ for a factor $c(t)$ set by the target and the path, and changing the variable from t to λ gives the form of Kingma and Gao (2023),

$$\mathcal{L}_w = \frac{1}{2}\int_{\lambda_{\min}}^{\lambda_{\max}} w(\lambda)\,\mathbb{E}\left\|\hat{\epsilon}_\theta(\mathbf{x}_\lambda,\lambda)-\epsilon\right\|^2 d\lambda, \qquad w(\lambda) = 2\,c(t)\,p(t)\left|\frac{dt}{d\lambda}\right|, \tag{17.51}$$

where $p(t)$ is the density of training times. The bound Equation 17.50 is the weighting $w \equiv 1$.

The expectation inside the integral does not depend on the path: at the same λ two paths hold the same noisy point up to scale (Section 17.2), so the network faces the same prediction problem. A schedule thus enters $\mathcal{L}_w$ only through the weighting and the endpoints $\lambda_{\min}$ and $\lambda_{\max}$; for the bound, schedules with the same endpoints give the same loss (Kingma et al. 2021).

As the smallest worked case, take the noise loss with uniform t on the linear path, $c = p = 1$. There $\sigma_t/\alpha_t = t/(1-t) = e^{-\lambda/2}$, so $t = 1/(1+e^{\lambda/2})$ and $|dt/d\lambda| = \frac{1}{8}\cosh^{-2}(\lambda/4)$. The weighting $w(\lambda) = \frac{1}{4}\cosh^{-2}(\lambda/4)$ is a bump centered at $\lambda = 0$, where signal and noise are equally strong, decaying like $e^{-|\lambda|/2}$ on both sides.

17.5.2.2 The Weighting Figure

The other objectives need two conversions. On the VP path the **v**-target of Salimans and Ho (2022) satisfies $\hat{\mathbf{v}} - \mathbf{v} = (\hat{\epsilon} - \epsilon)/\alpha_t$, so $c = 1/\alpha_t^2 = 1 + e^{-\lambda}$. The velocity on the linear path satisfies the same relation, and there $1/\alpha_t = 1 + e^{-\lambda/2}$. With uniform t this gives the following weightings, each up to a constant factor (4 derives them).

objective	path and noise levels	$w(\lambda)$
bound (ELBO)	any path	1
noise, DDPM	VP, linear β, t uniform	$\lvert dt/d\lambda\rvert$
noise	linear, t uniform	$\cosh^{-2}(\lambda/4)$, a bump at $\lambda = 0$
noise	cosine, t uniform	$\cosh^{-1}(\lambda/2)$
v	cosine, t uniform	$e^{-\lambda/2}$
velocity (flow matching)	linear, t uniform	$e^{-\lambda/2}$
EDM	$\log(\sigma_t/\alpha_t) \sim \mathcal{N}(-1.2, 1.2^2)$	$\mathcal{N}(\lambda; 2.4, 2.4^2)\,(e^{-\lambda}+\sigma_{\text{data}}^2)$

Two rows coincide: **v**-prediction on the cosine path and flow matching on the linear path are the same objective (Gao et al. 2025). The velocity and noise models of Section 17.2 differ in their weighting, not in what they learn (Figure 17.10).

Weightings that fall monotonically in λ are bounds themselves: such an $\mathcal{L}_w$ is the bound for data augmented with Gaussian noise of a random level (Kingma and Gao 2023), because a weighting that is one below some level μ and zero above is the bound Equation 17.50 for data noised to log-SNR μ, and a monotone weighting is a positive mixture of such steps. By this criterion **v**-prediction and flow matching are likelihood training of slightly noisy data; DDPM's noise loss and EDM's weighting, which both peak, are not.

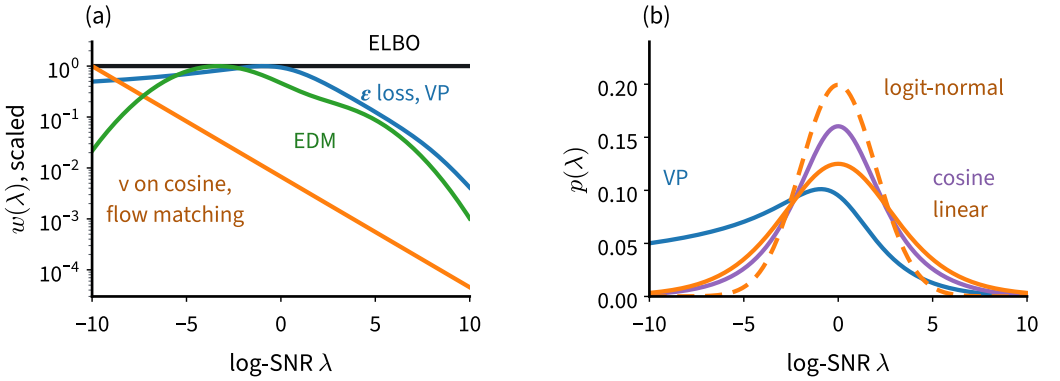


Figure 17.10: Diffusion objectives as weightings of the noise error over the log signal-to-noise ratio λ. (a) $w(\lambda)$ on a logarithmic scale: the bound is flat, the noise loss on the linear-β VP path peaks near $\lambda \approx -1$, **v**-prediction and flow matching share the curve $e^{-\lambda/2}$, and EDM's weighting ($\sigma_{\text{data}} = 0.5$) is a bump. (b) The density of λ under uniform t on the three paths, and under the logit-normal times of Esser et al. (2024) on the linear path.

17.5.2.3 The Time Distribution as Importance Sampling

Training times enter Equation 17.51 only through the product $w(\lambda) = 2c\,p\,|dt/d\lambda|$. In principle the two are separate: for λ drawn from any density $p(\lambda)$ that covers the range,

$$\mathcal{L}_w = \mathbb{E}_{\lambda\sim p}\Big[\frac{w(\lambda)}{2\,p(\lambda)}\,\mathbb{E}\big\|\hat{\epsilon}_\theta(\mathbf{x}_\lambda,\lambda)-\epsilon\big\|^2\Big],$$

the importance-sampling identity of Section 36.8.2, so $p(\lambda)$ changes only the variance of the estimate. Uniform t on the linear-β VP path spends almost half its draws below $\lambda = -3$ (panel (b) of Figure 17.10), where the bound's integrand is close to zero, and Nichol and Dhariwal (2021) reweighted them for this reason (2). Since in practice nobody divides by $p(\lambda)$, a time distribution such as the logit-normal of Esser et al. (2024) is a weighting in disguise.

17.5.3 The Design Space

A diffusion paper fixes a path, a target, a weighting, the training levels, a network and a sampler. Karras et al. (2022) separated these choices and showed that most of them can be made independently.

17.5.3.1 EDM's Table

EDM, as Karras et al. call their method, describes published methods by these columns:

method	training: path, levels, weighting	network output	sampler
DDPM	VP, linear β, t uniform, simple loss	noise	ancestral chain, 1,000 steps
NCSN	$\mathbf{x}_0 + \sigma\epsilon$, geometric ladder of σ, weight σ^2	score	annealed Langevin dynamics
flow matching	linear, t uniform	velocity	Euler or an adaptive ODE solver
EDM	$\log\bar{\sigma} \sim \mathcal{N}(-1.2, 1.2^2)$, weight $1/c_{\text{out}}^2$	preconditioned clean image	Heun on the $\rho = 7$ grid

NCSN (Song and Ermon 2019) (Section 38.5.3) never scales the data ($\alpha = 1$), the *variance-exploding* view. EDM works in these coordinates on every path, $\bar{\mathbf{x}} = \mathbf{x}/\alpha$ with noise level $\bar{\sigma} = \sigma/\alpha$, as `ode_sample` does.

EDM's new column is the parameterization. The network F shares its weights across noise levels, so it should see inputs and produce outputs of unit variance at every level. For data of standard deviation σ_{data}, the denoiser is

$$D(\bar{\mathbf{x}};\bar{\sigma}) = c_{\text{skip}}\,\bar{\mathbf{x}} + c_{\text{out}}\,F(c_{\text{in}}\,\bar{\mathbf{x}};\,\bar{\sigma}), \qquad c_{\text{in}} = \frac{1}{\sqrt{\bar{\sigma}^2+\sigma_{\text{data}}^2}}, \quad c_{\text{skip}} = \frac{\sigma_{\text{data}}^2}{\bar{\sigma}^2+\sigma_{\text{data}}^2}, \quad c_{\text{out}} = \frac{\bar{\sigma}\,\sigma_{\text{data}}}{\sqrt{\bar{\sigma}^2+\sigma} \tag{17.52}$$

and trains it on $\|D - \mathbf{x}_0\|^2$ with weight $1/c_{\text{out}}^2$, which gives F a unit-variance target too (10 derives the coefficients). At low noise the denoiser passes its input through ($c_{\text{skip}} \to 1$, $c_{\text{out}} \to 0$) and F only corrects it; at high noise the output is F's guess scaled to the data ($c_{\text{skip}} \to 0$, $c_{\text{out}} \to \sigma_{\text{data}}$).

`EDM` wraps a network $F(\mathbf{x}, t)$ in this preconditioning in the coordinates of a path ($c_{\text{in}}\bar{\mathbf{x}} = \mathbf{x}/n$ with $n^2 = \sigma^2 + \alpha^2\sigma_{\text{data}}^2$, finite even at $\alpha = 0$) and predicts the clean image; `edm_loss` draws the log-normal levels and applies the weight.

```
class EDM(nn.Module):
    """Karras et al.'s preconditioned denoiser on a path; returns x0-hat."""
    def __init__(self, net, path, sigma_data=0.5):
        super().__init__()
        self.net, self.path, self.sigma_data = net, path, sigma_data

    def forward(self, x, t):
        a = self.path.alpha(t).reshape(-1, 1, 1, 1)
        s = self.path.sigma(t).reshape(-1, 1, 1, 1)
        sd = self.sigma_data
        n = torch.sqrt(s ** 2 + (a * sd) ** 2)
        return a * sd ** 2 / n ** 2 * x + s * sd / n * self.net(x / n, t)

def edm_loss(edm, x0):
```

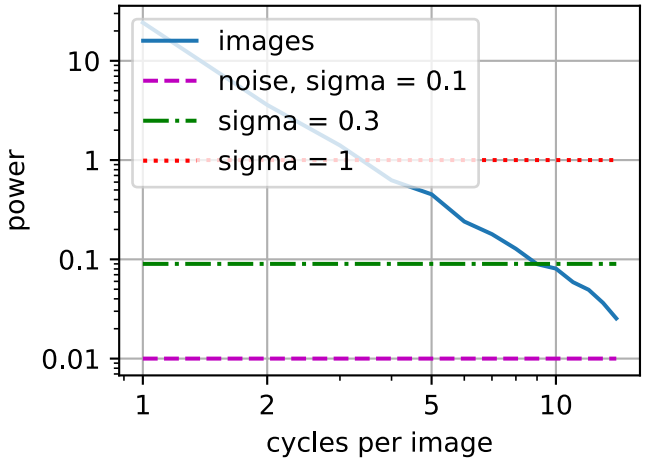
```
    """Log-normal noise levels sigma_bar, loss weight 1 / c_out^2."""
    sigma_bar = torch.exp(-1.2 + 1.2 * torch.randn(len(x0), device=x0.device))
    t = edm.path.time_at(-2 * torch.log(sigma_bar))
    x = edm.path.sample(x0, t, torch.randn_like(x0))
    sd = edm.sigma_data
    weight = (sigma_bar ** 2 + sd ** 2) / (sigma_bar * sd) ** 2
    return (weight.reshape(-1, 1, 1, 1) * (edm(x, t) - x0) ** 2).mean()
```

17.5.3.2 Noise Levels Are Scales

What does a noise level remove from an image? Under an orthonormal two-dimensional Fourier transform, white noise of standard deviation σ has power σ^2 at every frequency, while images concentrate their power at low frequencies; a frequency is hidden once the noise power exceeds the image's. The cell plots the power of the test images, averaged over rings of radius r cycles per image, against three noise levels.

```
X, y, X_test, _ = d2l.load_fashion_mnist_pm1()
F = torch.fft.fft2(X_test[:, 0], norm='ortho')    # noise has power sigma^2
k = torch.fft.fftfreq(28, 1 / 28, device=X.device)
ring = torch.sqrt(k[:, None] ** 2 + k ** 2).round().long().flatten()
power = (torch.bincount(ring, (F.abs() ** 2).mean(0).flatten())
         / torch.bincount(ring))                  # the mean over each ring
r, power = torch.arange(1, 15), power[1:15].cpu()
d2l.plot(r, [power] + [torch.full((14,), s ** 2) for s in (0.1, 0.3, 1.0)],
         'cycles per image', 'power', xscale='log', yscale='log',
         legend=['images', 'noise, sigma = 0.1', 'sigma = 0.3', 'sigma = 1'])
for s in (0.1, 0.3, 0.5, 1.0, 2.0):
    hidden = (power < s ** 2).nonzero()
    print(f'sigma = {s}: noise dominates from', f'{r[hidden[0]].item()} cycles'
          if len(hidden) else 'no ring below 15')
```

```
sigma = 0.1: noise dominates from no ring below 15
sigma = 0.3: noise dominates from 9 cycles
sigma = 0.5: noise dominates from 6 cycles
sigma = 1.0: noise dominates from 4 cycles
sigma = 2.0: noise dominates from 2 cycles
```



The image power falls by three orders of magnitude from one to fourteen cycles per image. Noise at $\sigma = 0.1$ stays below it everywhere, $\sigma = 0.3$ hides the rings from about 9 cycles per image outward, and $\sigma = 1$ from about 4. Adding noise is thus a low-pass filter whose cutoff falls as σ grows, and sampling runs from coarse to fine: the high-noise steps decide a garment's silhouette, the last steps its texture (Dieleman 2024).

The same view explains why a schedule depends on the resolution. At 56×56 with the same per-pixel noise, averaging each 2×2 block recovers the 28×28 image while the four noise values average to variance $\sigma^2/4$: for the coarse structure the signal-to-noise ratio has grown fourfold, and a schedule tuned at one resolution is too easy at a higher one. Hoogeboom et al. (2023) shift λ_t down by the logarithm of the ratio of pixel counts (6, Figure 17.11).

The VP path ends at $\lambda \approx -10$, so its network never trains on the pure noise the sampler starts from. At high resolution, where block averaging magnifies the leftover signal, this kept Stable Diffusion from generating very bright or very dark images (Lin, Liu, et al. 2024).

17.5.3.3 A Recipe Ablation

We compare four recipes on Fashion-MNIST with everything else fixed: noise and **v**-prediction on the VP path, the velocity on the linear path (the recipe of Section 17.3), and EDM on the VP path. Each trains an unconditional 64-channel `d2l.UNet` for 5,000 updates, a third of that section's budget, and samples the same 1,000 noise images with deterministic and stochastic DDIM ($\eta = 0$ and 1, 50 VP steps) and with 25 Heun steps on its own path, about 50 evaluations each. The Fréchet distance (FD) of Section 17.3 scores the samples against that of two sets of 1,000 test images.

```
cnn = d2l.fashion_feature_cnn(X, y)
f_test = d2l.fd_features(cnn, X_test[:2000])
floor = d2l.fid_score(f_test[:1000], f_test[1000:])
print(f'FD, real against real: {floor:.2f}')
```

```
FD, real against real: 0.89
```

A noise predictor cannot be queried at $\alpha = 0$, so DDIM runs every model on the VP grid, where α stays positive.

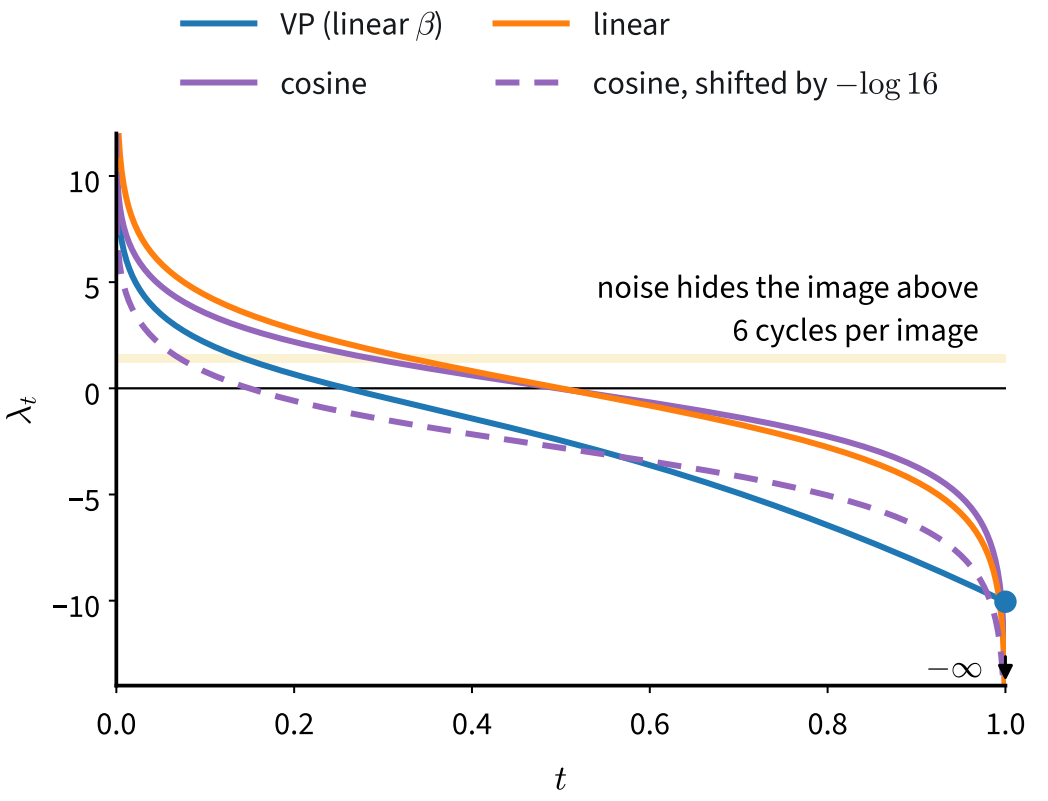


Figure 17.11: Noise schedules on the log-SNR clock: λ_t against t for the linear-β VP, cosine and linear paths, and the cosine path shifted for four times the side length ($\lambda_t - \log 16$). The VP path ends at $\lambda \approx -10$, the others at $-\infty$. The band marks the log-SNR at which noise hides the image above 6 cycles per image on Fashion-MNIST.

```
linear = d2l.GaussianPath.linear()
unet = lambda: d2l.UNet(ch=64)
recipes = {  # name: (path, kind of prediction, network, loss)
    'noise, vp': (vp, 'eps', unet, d2l.regression_loss(vp, 'eps')),
    'v, vp': (vp, 'v', unet, d2l.regression_loss(vp, 'v')),
    'velocity, linear': (linear, 'velocity', unet,
                         d2l.regression_loss(linear, 'velocity')),
    'EDM, vp': (vp, 'x0', lambda: EDM(unet(), vp), edm_loss)}
samplers = {  # name: sampler(pred, z, path); DDIM always on the VP time grid
    'DDIM-50, eta 0': lambda pred, z, path: d2l.ddim_sample(pred, z, vp, 50),
    'DDIM-50, eta 1': lambda pred, z, path: d2l.ddim_sample(pred, z, vp, 50,
                                                            1.0),
    'Heun-25': lambda pred, z, path: d2l.heun_sample(pred, z,
                                                     path.sigma_bars(25))}
```

The loop trains, samples and scores each model.

```
batch_fn = lambda: X[torch.randint(0, len(X), (128,), device=X.device)]
samples, fd = {}, {}
for name, (path, kind, make_net, loss_fn) in recipes.items():
    torch.manual_seed(0)
    _, ema_img, _ = d2l.train_regression(make_net().to(device), loss_fn,
                                         batch_fn, 5_000)
    pred = d2l.make_pred(ema_img, path, kind)
    for sampler, run in samplers.items():
        torch.manual_seed(1)                  # the same noise for every run
        z = torch.randn(1000, 1, 28, 28, device=device)
        samples[name, sampler] = run(pred, z, path)
        fd[name, sampler] = d2l.fid_score(
            d2l.fd_features(cnn, samples[name, sampler]), f_test[:1000])
print(f'{"FD":18}' + ''.join(f'{s:>16}' for s in samplers))
for name in recipes:
    print(f'{name:18}' + ''.join(f'{fd[name, s]:16.2f}' for s in samplers))
```

```
FD                 DDIM-50, eta 0  DDIM-50, eta 1         Heun-25
noise, vp                   12.85            9.30            9.99
v, vp                        8.85            8.72            7.69
velocity, linear             3.06            4.23            2.69
EDM, vp                      4.56            6.39            3.88
```

Two sets of real images score just under 1 against each other, and every entry is about three times that or more: 5,000 updates leave all four models far from the data. The recipes separate clearly. The velocity on the linear path scores best with every sampler, with EDM next; **v**-prediction on the VP path lands at two to three times the velocity model's FD with the deterministic samplers; and the noise predictor's deterministic samples are the worst entries, with stochastic DDIM better. Apart from the noise predictor, deterministic sampling is about as good as stochastic or better. Below are the noise predictor's samples (rows: DDIM with $\eta = 0$, $\eta = 1$, Heun).

```
rows = [samples['noise, vp', s][:10] for s in samplers]
d2l.show_gray(torch.cat(rows), 3, 10);
```



The deterministic samples are blotchy and over-saturated, the stochastic ones smoother garments. At $t = 1$ the VP path has $\alpha_1 \approx 0.007$, so $\hat{\mathbf{x}}_0 = (\mathbf{x} - \sigma\hat{\epsilon})/\alpha$ magnifies the noise predictor's error about 150-fold, and the noise loss gives these levels too little weight for a short run to make that error small. The velocity loss weights these levels heavily, and EDM's network outputs the clean image there without dividing. The **v**-loss, about $2 \cdot 10^4$ times the noise loss at $t = 1$, weights them heavily too, yet trails the velocity model, a gap that 4 probes. A stochastic step replaces part of the point with fresh noise and so corrects earlier errors, which a deterministic step carries to the end (Section 17.4).

17.5.3.4 Parameterization at Finite Capacity

For an unlimited network the four targets are one posterior mean in four forms, and choosing among them only chooses a weighting. A finite network finds some harder than others: an image lies near a set of much lower dimension than its pixel count, while the noise fills every dimension, so the clean image is the simpler target at every level. Li and He (2026) find that transformers on large patches fail at predicting the noise or the velocity and work at predicting the clean image.

17.5.4 Summary

Diffusion training is maximum likelihood up to a weighting. The chain's bound is a sum of noise regressions weighted by the drop in log-SNR per step, and on the mixture the trained network's bound and its probability-flow likelihood both come within about 0.01 nats of the true 2.5 nats. DDPM's simple loss, **v**-prediction, flow matching and EDM are the same integral with other weightings; the monotone ones are bounds for noise-augmented data. The parameterization decides what a finite network must compute, and the schedule matters through its endpoints and the resolution. Most of an image's pixels carry detail a viewer hardly sees: why diffuse the pixels at all, rather than a smaller representation?

17.5.5 Exercises

1. code **The prior term.** The reverse chain starts from $\mathcal{N}(\mathbf{0}, \mathbf{I})$, and L_T measures how far the end of the forward chain is from it.
   1. Approximating $\log(1 - \beta_k)$ by $-\beta_k$ gives $\alpha_T^2 \approx \exp(-\sum_k \beta_k)$. Evaluate the approximation for the VP path's chain at $T = 1000$ and compare it with the exact value. Which term of the expansion of $\log(1 - \beta_k)$ accounts for the difference?
   2. Use Equation 16.8 to write L_T as a function of α_T^2, d and $\|\mathbf{x}_0\|^2$, and show that it is approximately $\frac{1}{2}\alpha_T^2\|\mathbf{x}_0\|^2$ for small α_T^2. Compute $\mathbb{E}\|\mathbf{x}_0\|^2$ for the mixture from its parameters and evaluate L_T for vp and for a VP path with β from 0.1 to 5.

3. For the second path, draw 20,000 samples with `d2l.ancestral_sample` and the exact predictor, once starting from $\mathcal{N}(\mathbf{0}, \mathbf{I})$ and once from the exact $q(\mathbf{x}_T)$, which you obtain by noising draws of the mixture. Compare the outputs of `mix.summary`. Which statistic does the prior term affect? Compute the mean of $q(\mathbf{x}_T)$ and use it to explain the direction of the change. (The extended exercise "Prior mismatch, ODE versus SDE" of Section 38.4 compares how the reverse SDE and the probability-flow ODE carry such a mismatch to the data.)

2. code **The weights of the bound.** ⊠ Let w_k be the weight in Equation 17.49.
   1. Compute w_k for $\nu_k = \beta_k$ and for $\nu_k = \tilde{\beta}_k$ at every step $k = 2, \dots, 1000$ of the VP chain and plot both on a logarithmic axis. What fraction of $\sum_k w_k$ lies on the steps $k \le 20$ for each choice? What fraction does the simple loss put there?
   2. Evaluate the chain's bound Equation 17.44 on the mixture with the exact noise predictor, $\nu_k = \tilde{\beta}_k$, and the reconstruction term of the text at the noise level of step 1. Compare it with the true negative log-likelihood and with the continuous bound of the text. Show that over a step where the clean point is uncertain by about the step's own noise, the true reverse step has a variance larger than $\tilde{\beta}_k$, and estimate the divergence that each of the first steps contributes. Repeat with steps uniform in λ instead of t.
   3. A minibatch estimate of the chain's bound draws k uniformly. Nichol and Dhariwal (2021) (Section 3.3) instead draw k with probability p_k proportional to the root mean square of the per-sample value of L_{k-1} and divide the drawn term by p_k. Why does this reduce the variance of the estimate? Which steps would such a sampler draw most often on the mixture?

3. code **Target and weighting.** ⊠ Ho et al. (2020) (Table 2) compare predicting $\tilde{\mu}_k$ with predicting ϵ, each trained on the bound or on an unweighted squared error.
   1. Let the network output μ_θ directly and train it on Equation 17.46 with $\nu_k = \beta_k$. Solve Equation 17.48 for the implied noise prediction, and show that in terms of it the objective is the regression Equation 17.49 with the weights of the bound. What must a network that outputs μ_θ represent accurately, given that $\tilde{\mu}_k$ is nearly $\mathbf{x}_k$?
   2. With `TimeMLP` and the recipe of this section on the chain with $T = 1000$, train three models: noise prediction with the weights of the bound, mean prediction on the bound, and mean prediction on the unweighted error. Compare each with the model trained on the simple loss by the bound and the flow likelihood of the text, and by the summary statistics of DDIM samples with 50 steps. Which findings of Ho et al. (2020) does the comparison reproduce?

4. **The implied weighting.** Use Equation 17.51.
   1. Show that the velocity loss on the linear path with t uniform has $w(\lambda) \propto e^{-\lambda/2}$, and that the $\mathbf{v}$-loss on the cosine path $\alpha_t = \cos\frac{\pi t}{2}$, $\sigma_t = \sin\frac{\pi t}{2}$ has the same weighting.
   2. Show that the noise loss on the cosine path with t uniform has $w(\lambda) \propto \cosh^{-1}(\lambda/2)$, and compute w for the noise loss and the $\mathbf{v}$-loss on the linear-β VP path of the ablation.
   3. Which of these weightings are monotone in λ? Which noise levels does the noise loss on the VP path downweight relative to the $\mathbf{v}$-loss on the same path, and by what factor at its last step?
   4. (Code.) By the first part, the $\mathbf{v}$-loss on the cosine path weights the levels as the velocity loss does. Train it with the recipe of the ablation and score it with the three samplers. Does matching the weighting close the gap between $\mathbf{v}$-prediction on

the VP path and the velocity model? What else differs between the two recipes?

5. code **Likelihoods in many dimensions.** ⊠ In d dimensions the exact divergence in `pf_nll` takes d forward-mode derivatives per evaluation. Replace it by the estimate $\mathbf{r}^\top(\partial\mathbf{u}/\partial\mathbf{x})\,\mathbf{r}$ of Equation 38.21 with one Rademacher vector $\mathbf{r}$ per step, which takes one derivative (Grathwohl et al. 2019).
   1. Show that the resulting estimate of $\log p$ is unbiased. The variance of one probe is derived in Exercise 6a; measure the standard deviation of the estimate per point on the mixture, and compare it with the gap of a few thousandths of a nat between the trained network and the exact field. How many probes per step, or how many points, does resolving that gap take?
   2. Apply the estimate to the velocity model of the ablation on 100 test images, after adding uniform noise of the width of one gray level, $2/255$, to each pixel. Report the result in bits per dimension, $\big(-\log p(\mathbf{x})/d + \log 127.5\big)/\log 2$, which counts the bits per pixel of the 8-bit image. Compare it with the 8 bits of the raw encoding. What would 15,000 updates change?
6. **Resolution and the schedule.** Write $\mathrm{SNR}_t = \alpha_t^2/\sigma_t^2$ for data with unit variance per pixel.
   1. Show that multiplying the data by a constant κ multiplies the ratio of signal to noise variance in $\mathbf{x}_t$ by κ^2 at every t.
   2. Upsample each image to $28r \times 28r$ pixels by repeating every pixel in an $r \times r$ block, and noise the large image with the same schedule. Show that averaging the noisy image over the blocks gives the original image at the ratio $r^2\,\mathrm{SNR}_t$.
   3. For $r = 32$, an 896×896 image, compute $r^2\,\mathrm{SNR}_1$ for `vp`, and find the time of the 28×28 path with the same ratio. What does the result imply for an 896×896 model trained with this schedule, whose sampler starts from pure noise?
   4. Which shift of λ_t lets the averaged large image pass through the same ratios as the 28×28 model? Compare it with the shift of Hoogeboom et al. (2023) .
7. code **Paired likelihood comparisons.** Differences of hundredths of a nat between two models can only be resolved by pairing.
   1. For a fixed model p_θ and test points $\mathbf{x}_i$, let $d_i = \log p(\mathbf{x}_i) - \log p_\theta(\mathbf{x}_i)$. Show that the mean of the d_i is an unbiased estimate of $D_{\mathrm{KL}}(p \,\|\, p_\theta)$. Why can it be negative?
   2. Train a velocity model on the linear path with the recipe of this section. Compute the flow likelihood of both models on the same 4,000 test points, and report each model's mean negative log-likelihood with its standard error, then the mean of the per-point differences with its standard error. Which comparison resolves the two models? (Section 36.5.2.6 discusses the general case.)
   3. Do the same for the bound. Its log-SNR form is the same on every path, so for the velocity model draw λ, find the time of the linear path with `time_at`, and turn the velocity prediction into a noise prediction with `x0_eps`. Does the ordering of the two models agree with that of the flow likelihood?
8. **The forward posterior.** Derive Equation 17.45.
   1. Write $q(\mathbf{x}_{k-1} \mid \mathbf{x}_k, \mathbf{x}_0) \propto q(\mathbf{x}_k \mid \mathbf{x}_{k-1})\, q(\mathbf{x}_{k-1} \mid \mathbf{x}_0)$ as a product of two Gaussians in $\mathbf{x}_{k-1}$, add their precisions, and combine their means in proportion to precision.
   2. Show that $\tilde\beta_k \le \beta_k$, with equality only if $\sigma_{k-1}^2 = 1$. Where in the chain is $\tilde\beta_k$ much smaller than β_k?
   3. For $T = 2$ and $\beta_1 = \beta_2 = 1/2$, write out $\tilde\mu_2$ and $\tilde\beta_2$, and the three terms of Equation 17.44 for a network that predicts the noise exactly, for one-dimensional data $\mathcal{N}(0, s^2)$.

9. code **Clipping the predicted image.** The cosine path reaches $\alpha_1 = 0$.
   1. Train a noise predictor on `d2l.GaussianPath.cosine()` with the recipe of the ablation and sample it with `ddim_sample` on the same path. What happens at the first step, and why?
   2. Start the sampler instead at $t = 0.99$ from pure noise, with `t_start`. Compare the samples and their FD with those of the noise predictor of the ablation.
   3. Wrap the predictor so that it clips $\hat{\mathbf{x}}_0$ to $[-1, 1]$ and recomputes $\hat{\epsilon}$ from it, and sample again. Which part of the failure does clipping repair, and which does it not? Why does a $\mathbf{v}$-predictor on the same path need no clipping?
10. **EDM's preconditioning.** ⊠ Let $\mathbf{x}_0$ have zero mean and per-coordinate variance σ_{data}^2, independent of the noise.
    1. Choose c_{in} so that the input $c_{\text{in}}(\mathbf{x}_0 + \bar{\sigma}\epsilon)$ of F has unit variance. Writing the training target of F as $(\mathbf{x}_0 - c_{\text{skip}}\,\bar{\mathbf{x}})/c_{\text{out}}$, choose c_{skip} to minimize its variance, and then c_{out} to make that variance one. Check the result against Equation 17.52.
    2. Show that the loss $\|D - \mathbf{x}_0\|^2/c_{\text{out}}^2$ is $\|F - \text{target}\|^2$, and that, written as a noise loss, it is the noise error times $(e^{-\lambda} + \sigma_{\text{data}}^2)/\sigma_{\text{data}}^2$.
    3. With $\log\bar{\sigma} \sim \mathcal{N}(-1.2, 1.2^2)$, show that EDM's weighting is $w(\lambda) \propto \mathcal{N}(\lambda; 2.4, 2.4^2)\,(e^{-\lambda} + \sigma_{\text{data}}^2)$. Where is its peak? Kingma and Gao (2023) make it monotone by holding it at its peak value for all λ below the peak. By the theorem of the text, what does the change make of EDM's objective?

17.6 Latent Diffusion and Transformers

The image models of this chapter draw 28×28 garments with a U-Net that processes every pixel at each of fifty sampling steps. The text-to-image systems that many readers have used draw pictures of a million pixels in seconds, and the papers behind most of them describe neither a U-Net nor a network that looks at pixels. The previous section hinted at the reason: most pixels carry detail that a viewer hardly notices. Why do current systems diffuse something other than pixels, why is their network a transformer, and how does a caption steer it?

Rombach et al. (2022) split generation into two stages. An autoencoder compresses each image into a small grid of numbers, its *latent*, from which a decoder restores the image with little visible loss, and a diffusion model learns the distribution of the latents at a fraction of the cost per step. Peebles and Xie (2023) replaced the U-Net by a transformer that reads the latent grid as a sequence of patches, the *diffusion transformer* (DiT), and found that its samples improve steadily as its computation grows.

Together the two ideas are the design of most recent image and video generators. We train a small autoencoder on Fashion-MNIST and run the regression of the previous sections on its latents, then build the DiT and train it at three sizes, and close with text and video, a paragraph each. Systems at full scale are the subject of a later part of this book.

```
torch.set_float32_matmul_precision('high')  # tf32 matmuls on tensor cores
```

17.6.1 Why a Latent Space

17.6.1.1 The Cost of Pixels

Every sampling step evaluates the network on the whole image. A convolution applies its kernel at every pixel, so the cost of a U-Net grows with the number of pixels; attention compares every position with every other, so its cost grows with their square. A 512×512 color image holds $512 \cdot 512 \cdot 3 = 786{,}432$ numbers.

Most of these numbers describe detail. In Section 17.5.3.2 the power of Fashion-MNIST images fell by three orders of magnitude from one to fourteen cycles per image, and the sampler fixed the silhouette of a garment at high noise and added its texture in the last steps. A viewer rarely notices a wrong texture, but a missing sleeve is obvious.

Rate–distortion theory makes this precise: for each level of distortion, the rate–distortion function gives the fewest bits per image that a code needs (Section 37.6.3.2). Rombach et al. (2022) observed that a likelihood model of natural images spends most of its bits on imperceptible details, at the low-distortion end of this curve. They leave this *perceptual compression* to an autoencoder, and the *semantic compression*, the modeling of what the image shows, to the diffusion model. The autoencoder of Stable Diffusion maps a $512 \times 512 \times 3$ image to a $64 \times 64 \times 4$ latent, 48 times fewer numbers. Every diffusion step works on the latent, and the decoder runs once per sample.

The latent must keep enough information for the decoder and be easy for the diffusion model to learn. A latent that stored every pixel would reconstruct perfectly and be as hard to model as the pixels; Dieleman (2025) calls this a trade between rate, distortion and *modelability*. Our autoencoder maps 784 numbers to a 7×7 grid with four channels, 196 numbers (Figure 17.12).

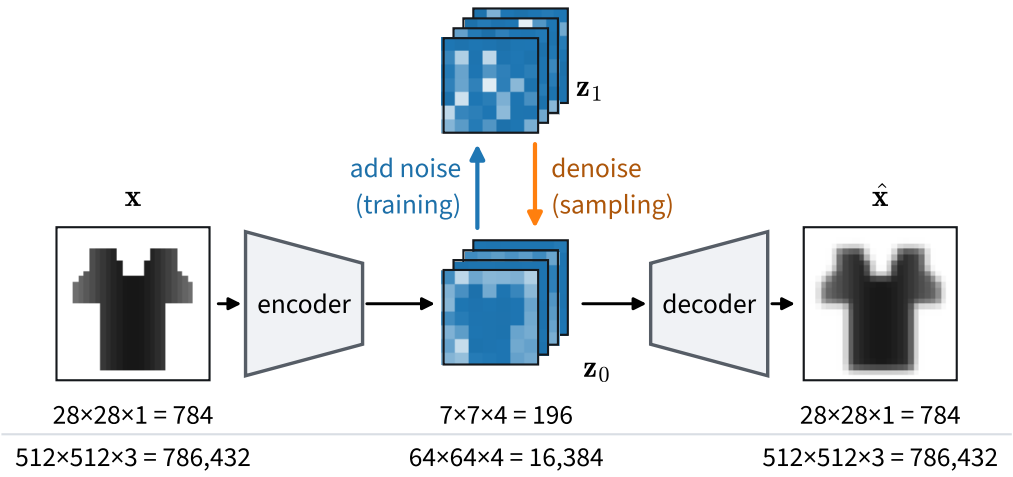


Figure 17.12: Latent diffusion. The encoder maps an image to a smaller latent, the diffusion model trains and samples on latents, and the decoder maps a sampled latent back to an image. Top: our autoencoder, from $28 \times 28 \times 1$ (784 numbers) to $7 \times 7 \times 4$ (196). Bottom: Stable Diffusion's, from $512 \times 512 \times 3$ (786,432) to $64 \times 64 \times 4$ (16,384).

17.6.1.2 The Autoencoder

A latent is easy to model if it has a known scale, against which the noise of the Gaussian path is calibrated, if it has no isolated outliers, and if nearby latents decode to similar images, so that a slightly wrong latent still gives a plausible image. An autoencoder trained only to reconstruct guarantees none of this.

The variational autoencoder (Kingma and Welling 2014) supplies the remedy. The encoder outputs a mean μ and a standard deviation $\mathbf{s}$ for each of the 196 latent numbers, the latent is drawn as $\mathbf{z} = \mu + \mathbf{s} \odot \xi$ with standard normal ξ, and the decoder maps $\mathbf{z}$ to a reconstruction $\hat{\mathbf{x}}$. The loss adds to the reconstruction error a term that pulls each latent number toward a standard normal:

$$\mathcal{L}(\mathbf{x}) = \frac{1}{784} \|\mathbf{x} - \hat{\mathbf{x}}\|^2 + \frac{\beta}{196} \sum_{j=1}^{196} \tfrac{1}{2} \left(\mu_j^2 + s_j^2 - 1 - \log s_j^2\right). \tag{17.53}$$

The sum is the Kullback–Leibler divergence from $\mathcal{N}(\mu, \operatorname{diag}(\mathbf{s}^2))$ to $\mathcal{N}(\mathbf{0}, \mathbf{I})$, the term of the evidence lower bound (Section 36.7), and β is its weight (not the schedule's β_k). With $\beta = 10^{-4}$ the reconstruction stays in charge, and the KL term only stops the encoder from inflating its means or shrinking its standard deviations to zero. The autoencoders of large systems add a patch critic and a perceptual loss, because a squared error averages plausible textures into a blur (Section 16.7); others quantize the latent into discrete tokens (Esser et al. 2021; Chang et al. 2022).

Our autoencoder stacks the residual blocks of the U-Net, without an embedding, at widths 64 and 128; its encoder halves the resolution twice to a 7×7 grid with a mean and a log variance per latent channel, and its decoder mirrors it.

```
class Autoencoder(nn.Module):
    """Residual convolutional autoencoder from 28x28x1 images to a 7x7
    Gaussian latent and back; returns the reconstruction and the KL term."""
    def __init__(self, latent_channels=4, ch=64, num_blocks=2):
        super().__init__()
        blocks = lambda a, b: [d2l.UNetBlock(a, b)] + [
            d2l.UNetBlock(b, b) for _ in range(num_blocks - 1)]
        down = lambda c: nn.Conv2d(c, c, 3, stride=2, padding=1)
        head = lambda a, b: [nn.GroupNorm(8, a), nn.SiLU(),
                             nn.Conv2d(a, b, 3, padding=1)]
        self.encoder = nn.Sequential(
            nn.Conv2d(1, ch, 3, padding=1), *blocks(ch, ch),
            down(ch), *blocks(ch, 2 * ch),                      # 14 x 14
            down(2 * ch), *blocks(2 * ch, 2 * ch),              # 7 x 7
            *head(2 * ch, 2 * latent_channels))  # mean and log variance
        self.decoder = nn.Sequential(
            nn.Conv2d(latent_channels, 2 * ch, 3, padding=1),
            *blocks(2 * ch, 2 * ch),
            nn.Upsample(scale_factor=2), *blocks(2 * ch, ch),   # 14 x 14
            nn.Upsample(scale_factor=2), *blocks(ch, ch),       # 28 x 28
            *head(ch, 1))

    def encode(self, x):
```

```
        mean, log_var = self.encoder(x).chunk(2, dim=1)
        return mean, log_var.clamp(-30, 20)

    def forward(self, x):
        mean, log_var = self.encode(x)
        z = mean + (0.5 * log_var).exp() * torch.randn_like(mean)
        kl = 0.5 * (mean ** 2 + log_var.exp() - 1 - log_var).mean()
        return self.decoder(z), kl
```

We train it for 20,000 updates with `d2l.train_regression` of Section 17.1, which runs Adam on a loss and returns the trained model, a moving average of its weights and the losses. The learning rate decays to zero along a cosine, so the final weights need no averaging.

```
X, y, X_test, y_test = d2l.load_fashion_mnist_pm1()
dev = X.device

def batch_fn():
    return X[torch.randint(0, len(X), (128,), device=dev)]

def ae_loss(ae, x):
    x_hat, kl = ae(x)
    return ((x_hat - x) ** 2).mean() + 1e-4 * kl

torch.manual_seed(1)
ae, _, _ = d2l.train_regression(d2l.Autoencoder().to(dev), ae_loss, batch_fn,
                                20_000, cosine=True)
print(f'{d2l.num_params(ae) / 1e6:.2f}M parameters')
```

```
2.40M parameters
```

The first row shows eight test images, the second the decoder's output for their latent means.

```
with torch.no_grad():
    x_hat = ae.decoder(ae.encode(X_test[:8])[0])
d2l.show_gray(torch.cat([X_test[:8], x_hat]), 2, 8);
```



The reconstructions are hard to tell from the originals. To measure them we use the Fréchet distance (FD) of Section 17.3, the distance between Gaussians fitted to the features that a small classifier, `d2l.FeatureCNN`, extracts from two sets of images. Two sets of real images differ by chance too, so we print the FD between two disjoint sets of 5,000 test images as the *floor*. All FDs in this section use 5,000 images; at 1,000 the floor is about six times higher.

```
cnn = d2l.fashion_feature_cnn(X, y)
f_ref = d2l.fd_features(cnn, X_test[:5000])

def fd(imgs):
    """FD of images against the first 5,000 test images."""
    return d2l.fid_score(d2l.fd_features(cnn, imgs), f_ref)

print(f'floor, the other 5,000 test images: FD {fd(X_test[5000:]):.2f}')
```

```
floor, the other 5,000 test images: FD 0.17
```

We score the reconstructions of the other 5,000 test images.

```
x_hat = d2l.batched(lambda x: ae.decoder(ae.encode(x)[0]), X_test[5000:])
fd_rec = fd(x_hat)
print(f'reconstructions: squared error {((x_hat - X_test[5000:]) ** 2).mean():.4f}'
      f' per pixel, FD {fd_rec:.2f}')
```

```
reconstructions: squared error 0.0038 per pixel, FD 0.63
```

The reconstructions score an FD of 0.6 to 0.8 against the floor of 0.17, at a squared error of about 0.004 per pixel. This bounds every latent model trained with this autoencoder: perfect latents would decode to reconstructions and score the same.

We encode the training set once and print the standard deviation of the latents, the average of the encoder's $\mathbf{s}$, and the excess kurtosis of all latent values, which is zero for a normal distribution and large when a few values lie far out.

```
encoded = d2l.batched(lambda x: torch.cat(ae.encode(x), 1), X)
mu, log_var = encoded.chunk(2, dim=1)       # mean and log variance
s = (0.5 * log_var).exp()
z = mu + s * torch.randn_like(mu)
scale = 1 / z.std().item()
kurtosis = (((z - z.mean()) * scale) ** 4).mean() - 3
print(f'latent std {z.std():.2f} (scale factor {scale:.2f}), encoder std '
      f'{s.mean():.3f}, excess kurtosis {kurtosis:.2f}')
```

```
latent std 0.94 (scale factor 1.07), encoder std 0.026, excess kurtosis -0.20
```

The latents have a standard deviation slightly below one, so we multiply them by a factor c of about 1.05. The encoder's standard deviation is 0.02 to 0.03, so the KL term barely blurs each code, and the excess kurtosis is near zero or negative: the latents have no heavy tails. A larger β makes them closer to normal but costs reconstruction quality (7).

17.6.2 Diffusion in the Latent Space

17.6.2.1 The Same Regression on Latents

With the autoencoder fixed, nothing changes but the data. Each batch draws the latents of 128 training images, $\mathbf{z}_0 = c\,(\mu + \mathbf{s} \odot \xi)$, fresh from the encoder's distribution as Rombach et al. (2022) do. We train on the linear path of Section 17.2: we form $\mathbf{z}_t = (1-t)\,\mathbf{z}_0 + t\,\epsilon$ and regress the network's output onto the velocity $\epsilon - \mathbf{z}_0$.

The scale factor matters because the path mixes the data with noise of unit variance. Latents with a standard deviation of five would shift the log-SNR at every time by $\log 25$ (1), and Stable Diffusion multiplies its latents by 0.18215 for this reason. The network is the U-Net of Section 17.3 with four input and output channels, and one `loss_fn` serves images and latents.

```
path = d2l.GaussianPath.linear()
loss_fn = d2l.regression_loss(path, 'velocity')  # for images or latents

def latent_batch_fn():
    i = torch.randint(0, len(mu), (128,), device=dev)
    return (mu[i] + s[i] * torch.randn_like(mu[i])) * scale

torch.manual_seed(3)
_, unet_latent, _ = d2l.train_regression(
    d2l.UNet(ch=64, in_channels=4, out_channels=4).to(dev), loss_fn,
    latent_batch_fn, 20_000)
```

As a reference we train the same U-Net on pixels. Both runs take 20,000 updates, twice the budget of Section 17.4, because the transformers below need longer runs and this pixel model is their reference too.

```
torch.manual_seed(4)
_, unet_pixel, _ = d2l.train_regression(d2l.UNet(ch=64).to(dev), loss_fn,
                                        batch_fn, 20_000)
```

`sample` draws 5,000 samples from fixed noise with 50 Euler steps of `d2l.ode_sample` (Section 17.2), and `decode` undoes the scale factor and decodes latents. The first row below holds samples of the pixel model, the second decoded samples of the latent model.

```
def sample(net, shape, n=5000):
    """n samples by 50 Euler steps from fixed noise, 1,000 at a time."""
    gen = torch.Generator(dev).manual_seed(0)
    noise = torch.randn(n, *shape, device=dev, generator=gen)
    pred = d2l.make_pred(net, path, 'velocity')
    return d2l.batched(lambda x: d2l.ode_sample(pred, x, path.sigma_bars(50)),
                       noise)

def decode(z):
    return d2l.batched(lambda z_k: ae.decoder(z_k / scale), z)

latent_shape, image_shape = (4, 7, 7), (1, 28, 28)
x_pixel = sample(unet_pixel, image_shape)
x_latent = decode(sample(unet_latent, latent_shape))
fd_pixel, fd_latent = fd(x_pixel), fd(x_latent)
print(f'FD: pixel U-Net {fd_pixel:.2f}, latent U-Net {fd_latent:.2f}')
d2l.show_gray(torch.cat([x_pixel[:8], x_latent[:8]]), 2, 8);
```

```
FD: pixel U-Net 1.50, latent U-Net 2.07
```



Both rows show recognizable garments. The latent model scores an FD of about 2 (2.0 to 2.4), the pixel model 1.5 to 1.6.

What did the latent model save? `train_step_gflops` counts the floating-point operations of one training step, forward and backward on 128 inputs, in billions (GFLOP).

```
def train_step_gflops(net, loss_fn, shape, batch_size=128):
    """GFLOP of one training step, forward and backward, on a batch of zeros
    of the given shape."""
    from torch.utils.flop_counter import FlopCounterMode  # not in the preamble
    x = torch.zeros(batch_size, *shape, device=next(net.parameters()).device)
    with FlopCounterMode(display=False) as counter:
        loss_fn(net, x).backward()
    return counter.get_total_flops() / 1e9
```

```
gflops_latent = d2l.train_step_gflops(
    d2l.UNet(ch=64, in_channels=4, out_channels=4).to(dev), loss_fn,
    latent_shape)
gflops_pixel = d2l.train_step_gflops(d2l.UNet(ch=64).to(dev), loss_fn,
                                     image_shape)
print(f'GFLOP per training step: latent U-Net {gflops_latent:.0f}, '
      f'pixel U-Net {gflops_pixel:.0f}')
```

```
GFLOP per training step: latent U-Net 15, pixel U-Net 203
```

The latent U-Net needs about a dozen GFLOP per step, the pixel U-Net about 200. Every evaluation of the latent network, in training and at each sampling step, costs less than a tenth of the arithmetic.

17.6.2.2 Reading the Result

At equal updates the latent model's FD is about 1.4 to 1.5 times the pixel model's. It cannot fall below the reconstruction FD of 0.6 to 0.8, and it sits well above it. The gap combines the latent model's own error with the decoder's response to latents that no training image produced.

At 28×28 the trade is not attractive: the pixel model is cheap anyway, and the autoencoder needs a training run of its own. At 512×512 the factor of 48 applies to every convolution, and for a transformer with 2×2 patches the number of tokens falls from 65,536 to 1,024, so attention, whose cost grows with their square, becomes 4,096 times cheaper. This is why most large text-to-image systems since 2022 diffuse latents, and why their autoencoder limits their quality, a limit we measure once we have a second network to try.

17.6.3 The Transformer Backbone

The U-Net builds in the assumptions of a convolutional network: nearby pixels matter most, and the same features can appear anywhere. The vision transformer of Section 11.5 makes fewer of them, and on image classification it trails convolutional networks at small budgets and overtakes them at large ones. Peebles and Xie (2023) asked whether the same holds for diffusion.

17.6.3.1 Patches and Tokens

The DiT reads its input as the vision transformer does. It cuts the image into $p \times p$ patches, maps each to a vector of width d by a linear map (a convolution with kernel size and stride p), adds a learned position embedding, and passes the tokens through pre-norm transformer blocks (Section 11.1). With $p = 4$ a 28×28 image becomes 49 tokens of 16 pixels each; on our latents with $p = 1$, each of the 49 grid positions is a token of four numbers.

The output differs from a classifier's. A denoiser returns a prediction of the shape of its input, so the DiT maps every token by a final linear layer to the values of its patch and reassembles the patches. Each token predicts the velocity of its own patch, informed through attention by all the others. With one token per pixel the image would have 16 times as many tokens, and attention would cost 256 times as much.

17.6.3.2 Adaptive Layer Norm

The transformer also needs the time t, and possibly a label. Peebles and Xie (2023) compared ways to condition it, among them appending the condition as extra tokens, and found that the cheapest worked best: the condition sets the normalization.

A layer norm standardizes each token and applies a learned scale and shift. In *adaptive layer norm* a linear map computes the scale $\mathbf{a}$ and the shift $\mathbf{b}$ from the condition $\mathbf{c}$, the sum of a time embedding and a label embedding, and the condition also sets a *gate* $\mathbf{g}$ on each sublayer's output. One block computes

$$\begin{aligned}\mathbf{X} &\leftarrow \mathbf{X} + \mathbf{g}_1 \odot \text{Attention}\big(\text{LN}(\mathbf{X}) \odot (1 + \mathbf{a}_1) + \mathbf{b}_1\big),\\ \mathbf{X} &\leftarrow \mathbf{X} + \mathbf{g}_2 \odot \text{MLP}\big(\text{LN}(\mathbf{X}) \odot (1 + \mathbf{a}_2) + \mathbf{b}_2\big),\end{aligned}$$

where LN has no scale or shift of its own and the six vectors are one linear function of $\mathbf{c}$, shared by all tokens of an image. One small matrix product per image thus turns each feature of each sublayer up or down.

The linear map starts at zero, hence the name *adaLN-Zero*. All gates are then zero, every block computes the identity, and the final layer, also zero, outputs zero. Training starts from a network that predicts zero velocity everywhere, and each block comes into play as its gates grow (Figure 17.13).

`DiTBlock` builds the block on the `ViTBlock` of Section 11.5, with `ada` as the linear map.

```
class DiTBlock(d2l.ViTBlock):
    """ViTBlock with adaLN-Zero: the condition c sets a shift, a scale and a
    gate for each sublayer; the gates start at zero."""
    def __init__(self, num_hiddens, mlp_num_hiddens, num_heads):
```

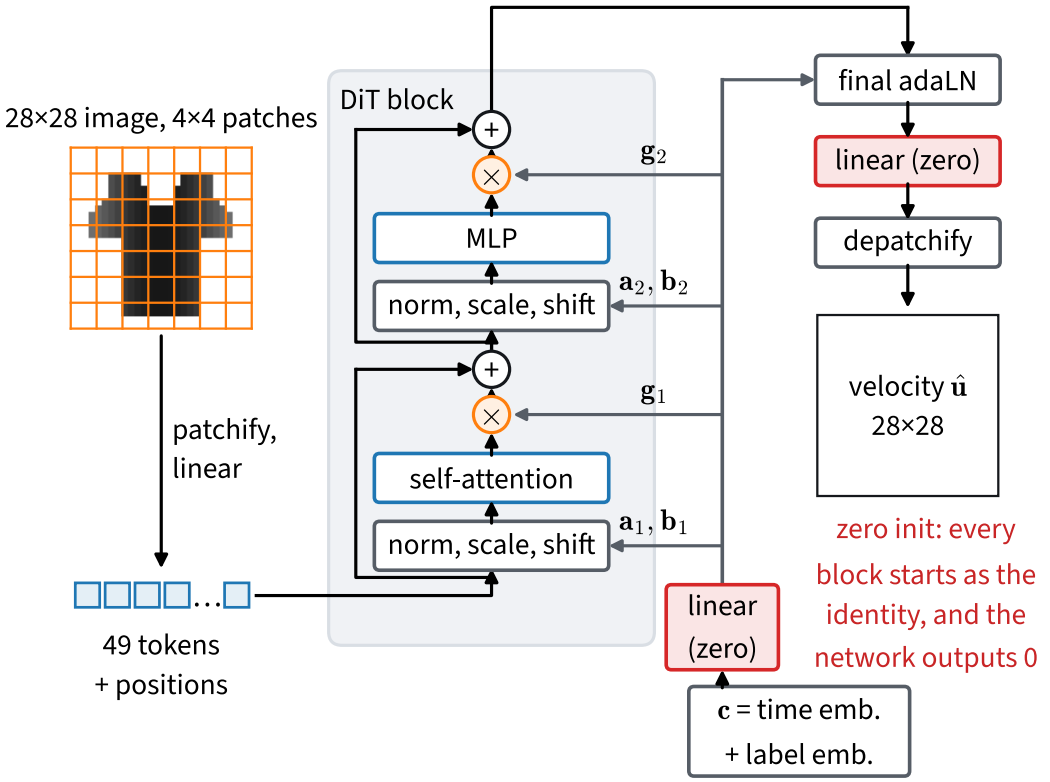


Figure 17.13: The diffusion transformer. Left: 4×4 patches become 49 tokens with learned positions. Middle: a DiT block; the condition $\mathbf{c}$, the sum of the time and label embeddings, sets a shift, a scale and a gate for each sublayer, and the gates start at zero. Right: a final adaptive layer norm and a zero-initialized linear layer map each token to its patch.

```
        super().__init__(num_hiddens, mlp_num_hiddens, num_heads, dropout=0.0,
                         use_bias=True)
        ln = lambda: nn.LayerNorm(num_hiddens, elementwise_affine=False,
                                  eps=1e-6)
        self.norm1, self.norm2 = ln(), ln()
        self.ada = nn.Sequential(nn.SiLU(),
                                 nn.Linear(num_hiddens, 6 * num_hiddens))
        nn.init.zeros_(self.ada[1].weight), nn.init.zeros_(self.ada[1].bias)

    def forward(self, X, c):
        shift1, scale1, gate1, shift2, scale2, gate2 = (
            self.ada(c)[:, None].chunk(6, dim=-1))
        X_mod = self.norm1(X) * (1 + scale1) + shift1
        X = X + gate1 * self.attention(X_mod, X_mod, X_mod)
        return X + gate2 * self.ffn(self.norm2(X) * (1 + scale2) + shift2)
```

`DiT` adds the patch embedding, the learned positions, the time and label embeddings, and the final adaptive layer norm and linear layer.

```
class DiT(nn.Module):
    """Diffusion transformer conditioned on a time t in [0, 1] and, with
    num_classes, on a label y; label num_classes means no label."""
    def __init__(self, img_size=28, patch_size=4, num_channels=1, width=256,
                 depth=6, num_heads=4, num_classes=None):
        super().__init__()
        self.width, self.num_classes = width, num_classes
        self.p, self.g = patch_size, img_size // patch_size
        self.patch = d2l.PatchEmbedding(img_size, patch_size, width,
                                        num_channels)
        self.pos = nn.Parameter(0.02 * torch.randn(1, self.g ** 2, width))
        self.time_mlp = d2l.make_mlp([width] * 3, nn.SiLU())
        self.label_emb = (nn.Embedding(num_classes + 1, width)
                          if num_classes else None)
        self.blks = nn.ModuleList([DiTBlock(width, 4 * width, num_heads)
                                   for _ in range(depth)])
        self.ln = nn.LayerNorm(width, elementwise_affine=False, eps=1e-6)
        self.ada = nn.Sequential(nn.SiLU(), nn.Linear(width, 2 * width))
        self.out = nn.Linear(width, patch_size ** 2 * num_channels)
        for m in (self.ada[1], self.out):
            nn.init.zeros_(m.weight), nn.init.zeros_(m.bias)

    def forward(self, x, t, y=None):
        c = self.time_mlp(d2l.time_embedding(t, self.width))
        if y is not None:
            c = c + self.label_emb(y)
        X = self.patch(x) + self.pos
        for blk in self.blks:
            X = blk(X, c)
        shift, scale = self.ada(c)[:, None].chunk(2, dim=-1)
        X = self.out(self.ln(X) * (1 + scale) + shift)  # (n, g*g, p*p*c)
        n, g, p = len(x), self.g, self.p  # tokens back to pixels
        X = X.reshape(n, g, g, p, p, -1).permute(0, 5, 1, 3, 2, 4)
```

```
        return X.reshape(n, -1, g * p, g * p)
```

We build a DiT of width 256 and depth 6 with four heads, and pass eight images and times through it.

```
dit = d2l.DiT().to(dev)       # width 256, depth 6, 4 heads, 4 x 4 patches
x, t = X[:8], torch.rand(8, device=dev)
with torch.no_grad():
    tokens, out = dit.patch(x), dit(x, t)
print('images', tuple(x.shape), '-> tokens', tuple(tokens.shape),
      '-> output', tuple(out.shape))
print(f'parameters: DiT {d2l.num_params(dit) / 1e6:.2f}M, '
      f'U-Net {d2l.num_params(d2l.UNet(ch=64)) / 1e6:.2f}M')
print('output at initialization all zero:', bool((out == 0).all()))
```

```
images (8, 1, 28, 28) -> tokens (8, 49, 256) -> output (8, 1, 28, 28)
parameters: DiT 7.39M, U-Net 1.20M
output at initialization all zero: True
```

The DiT has 7.39 million parameters, six times as many as the U-Net, and its untrained output is exactly zero.

17.6.3.3 Scaling with Compute

Is the transformer better than the U-Net? At one size the question has no general answer. Peebles and Xie (2023) trained DiTs of four sizes with three patch sizes each. At a fixed number of training steps, FID fell steadily as the network's arithmetic per forward pass grew, whether it came from a larger network or from smaller patches, and their largest model, trained for seven million steps, set a new best FID on class-conditional ImageNet at 256×256.

We repeat the measurement at our scale. Three DiTs grow in width and depth together, with 64 dimensions per head as in the paper: DiT-S (width 128, depth 4, two heads), DiT-M (the model above) and DiT-L (width 384, depth 8, six heads). Each is trained on pixels with the U-Net's loss and recipe for the same 20,000 updates, and we measure its size by the FLOPs per training step.

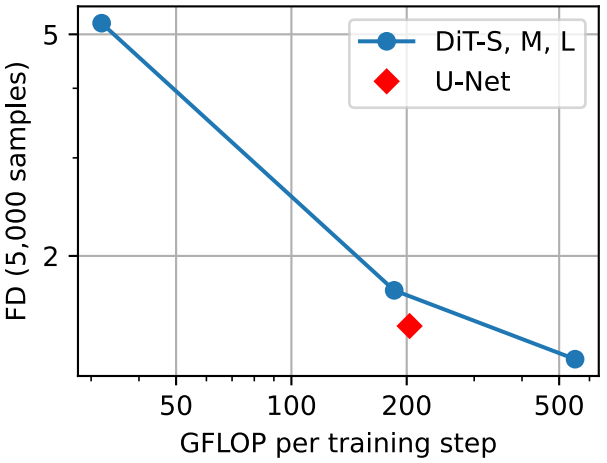
```
sizes = {'DiT-S': dict(width=128, depth=4, num_heads=2),
         'DiT-M': dict(width=256, depth=6, num_heads=4),
         'DiT-L': dict(width=384, depth=8, num_heads=6)}
runs = {}
for seed, (name, size) in enumerate(sizes.items(), start=5):
    gflops = d2l.train_step_gflops(d2l.DiT(**size).to(dev), loss_fn,
                                   image_shape)
    torch.manual_seed(seed)
    net = d2l.DiT(**size).to(dev)
    _, ema, _ = d2l.train_regression(net, loss_fn, batch_fn, 20_000)
    imgs = sample(ema, image_shape)
    runs[name] = dict(params=d2l.num_params(net), gflops=gflops, fd=fd(imgs),
                      imgs=imgs[:8])
runs['U-Net'] = dict(params=d2l.num_params(d2l.UNet(ch=64)),
```

```
                    gflops=gflops_pixel, fd=fd_pixel, imgs=x_pixel[:8])
for name, run in runs.items():
    print(f'{name:6} {run["params"] / 1e6:5.2f}M parameters '
          f'{run["gflops"]:4.0f} GFLOP per step   FD {run["fd"]:.2f}')
```

```
DiT-S   1.26M parameters   32 GFLOP per step   FD 5.24
DiT-M   7.39M parameters  185 GFLOP per step   FD 1.73
DiT-L  21.90M parameters  550 GFLOP per step   FD 1.31
U-Net   1.20M parameters  203 GFLOP per step   FD 1.50
```

We plot FD against FLOPs per step on logarithmic axes.

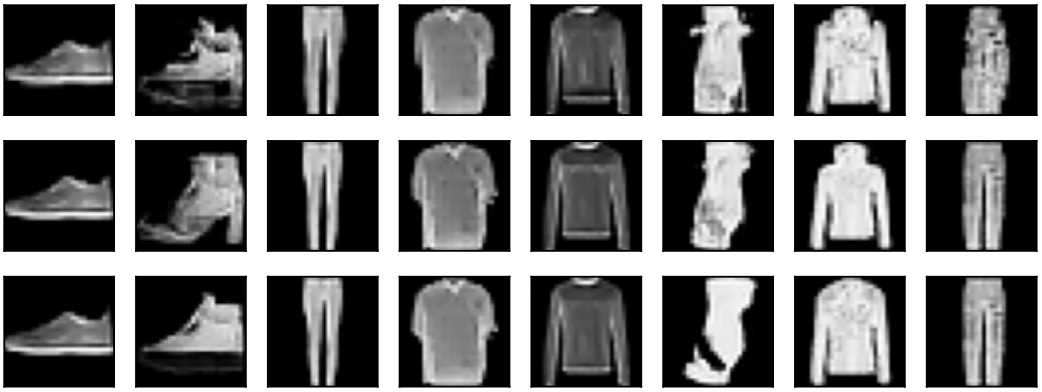
```
dits = [runs[name] for name in sizes]
d2l.plot([[run['gflops'] for run in dits], [runs['U-Net']['gflops']]],
         [[run['fd'] for run in dits], [runs['U-Net']['fd']]],
         'GFLOP per training step', 'FD (5,000 samples)',
         legend=['DiT-S, M, L', 'U-Net'], xscale='log', yscale='log',
         fmts=('o-', 'rD'))
```



FD falls with every step up in size: about 5 for DiT-S at about 30 GFLOP per step, 1.5 to 1.7 for DiT-M at about 190, and 1.2 to 1.5 for DiT-L at about 550. The first sixfold increase in compute divides the FD by about three; the next threefold increase gains less. The U-Net, at 1.5 to 1.6 and about 200 GFLOP, matches or slightly beats DiT-M, which spends the same arithmetic with six times the parameters, and DiT-L beats it.

The rows below show the first eight samples of DiT-S, DiT-M and DiT-L, drawn from the same noise.

```
d2l.show_gray(torch.cat([run['imgs'] for run in dits]), 3, 8);
```



The three models mostly draw the same garment from the same noise, since each integrates an ODE that sends a noise image to one image. DiT-S draws it with blotches and broken outlines, and each larger model sharpens it.

At equal parameters the U-Net wins by a wide margin: DiT-S has as many parameters and three times the FD. The convolutional assumptions are worth much on a small budget, and the transformer overtakes the U-Net only by growing. What makes it the backbone at scale is that it keeps improving as it grows, by a recipe as simple as making it wider and deeper. Its arithmetic runs as large matrix products, and the methods that train language models at scale carry over. Our horizontal axis is compute per update. Whether a large model trained for fewer updates beats a small one trained for more, at equal total compute, is a separate question (11).

Does the transformer help in the latent space too? We train DiT-M on the latents with patches of size 1, so that it sees 49 tokens and spends the same arithmetic per step as in pixel space.

```
gflops = d2l.train_step_gflops(
    d2l.DiT(img_size=7, patch_size=1, num_channels=4, **sizes['DiT-M']).to(dev),
    loss_fn, latent_shape)
torch.manual_seed(8)
net = d2l.DiT(img_size=7, patch_size=1, num_channels=4,
              **sizes['DiT-M']).to(dev)
_, dit_latent, _ = d2l.train_regression(net, loss_fn, latent_batch_fn, 20_000)
print(f'latent DiT-M, {gflops:.0f} GFLOP per step: FD '
      f'{fd(decode(sample(dit_latent, latent_shape))):.2f}; latent U-Net '
      f'{fd_latent:.2f}; reconstructions {fd_rec:.2f}')
```

```
latent DiT-M, 185 GFLOP per step: FD 1.97; latent U-Net 2.07; reconstructions 0.63
```

The latent DiT-M scores an FD of about 2 (1.8 to 2.3), at or a little below the latent U-Net's, although in pixel space it reached 1.5 to 1.7. In the latent space the backbone matters little: both land at about three times the reconstruction FD. Both backbones, a dozen and two hundred GFLOP per step, land at about the same FD, so the limit lies with the autoencoder, its reconstructions and its decoder's response to slightly wrong latents (10). At our scale the best pixel model therefore beats every latent one. At 512×512 the saving decides, but the limit remains, which is why the design of image autoencoders, often called *tokenizers*, has become a field of its own.

17.6.4 Conditioning on Text and Beyond

17.6.4.1 Cross-Attention and Joint Attention

A caption enters the model as a sequence of vectors. A frozen text encoder, such as the text half of CLIP (Radford et al. 2021) or the encoder of T5 (Raffel et al. 2020), turns the caption into one vector per token, and the denoiser attends to them. In the U-Net of Rombach et al. (2022), cross-attention layers let the image positions query the caption vectors as keys and values. Esser et al. (2024) instead concatenate image and caption tokens into one sequence and attend over both, with separate weights for each kind of token; this *multimodal DiT* (MMDiT) is the backbone of Stable Diffusion 3. Training needs one ingredient this chapter already has. The caption is dropped with a small probability, as the label was in Section 17.3.4.2, so the empty caption plays the null label $\emptyset$, and classifier-free guidance combines the two predictions as before, $\tilde{\mathbf{u}} = \mathbf{u}_\theta(\mathbf{x}, t, \emptyset) + \gamma\,(\mathbf{u}_\theta(\mathbf{x}, t, c) - \mathbf{u}_\theta(\mathbf{x}, t, \emptyset))$, with c now the caption. Esser et al. also draw the training times from the logit-normal distribution of Section 17.5.

17.6.4.2 Video

A video is a stack of frames, and the recipe applies once the stack is compressed. A three-dimensional autoencoder compresses space and time together, typically eightfold along each side of a frame and fourfold or more in time, and the transformer attends over patches that span a few frames each. Ho et al. (2022) trained the first video diffusion models on pixels, Blattmann et al. (2023) adapted a latent image model to video, and Polyak et al. (2024) trained a transformer of 30 billion parameters on video latents. The number of tokens sets the cost: a few seconds of video make tens of thousands of tokens, and attention over all of them dominates training.

17.6.5 Summary

Diffusion does not need to run on pixels. An autoencoder with a small KL term compresses each image into a latent of known scale, the regression of the previous sections runs on the latents unchanged, and the decoder turns sampled latents back into images. Each network evaluation then costs a fraction of a pixel model's, while the reconstructions of the autoencoder limit the quality of the samples. The diffusion transformer reads latents or pixels as patch tokens and lets the time set a scale, a shift and a gate in every block, starting from the identity. At equal updates its FD fell steadily with its compute per step, and its largest version beat the U-Net. A caption enters as tokens through attention, and video as tokens over space and time. However cheap each evaluation is, fifty evaluations per sample remain fifty; the next section asks how to get by with one or two.

17.6.6 Exercises

1. **The scale of the latents.** The linear path mixes the data with noise of unit variance.
    1. Suppose each latent value has standard deviation ς instead of one. Show that its signal-to-noise ratio at time t is $\varsigma^2(1-t)^2/t^2$, so that the log-SNR is shifted by $\log \varsigma^2$ at every time.
    2. At which time does a latent with $\varsigma = 5$ have the signal-to-noise ratio that a latent with $\varsigma = 1$ has at $t = \frac{1}{2}$? What fraction of uniformly drawn training times then falls where the signal still dominates the noise?

3. code Train the latent U-Net on latents multiplied by $5c$ instead of c, and compare the decoded FD. Relate the result to the shift of the schedule with resolution in Section 17.5.3.2.

2. **Tokens at high resolution.** The autoencoder of Stable Diffusion maps a $512 \times 512 \times 3$ image to $64 \times 64 \times 4$.
   1. How many tokens does a DiT see with patches of size 2 on the latent, and with patches of size 16 on the pixels? How many numbers does each token carry in the two cases?
   2. A DiT block of width d has about $12d^2$ weights, and a training step costs about six FLOPs per weight and token, plus the attention. Estimate the FLOPs of one training step on a batch of 256 for a DiT of width 1,152 and depth 28 on the latents with patch 2. How large is the attention part?
   3. With patch 16 on the pixels the DiT sees as many tokens as with patch 2 on the latents, at almost the same cost. What does the autoencoder provide that a linear patch embedding does not?
3. code **Why start at zero.** Replace the zero initialization of the `ada` layers in `d2l.DiTBlock` and `d2l.DiT` by the framework's default initialization.
   1. Print the size of the output of the untrained network for both initializations.
   2. Train both DiT-M models for 2,000 updates and plot their losses. Explain the difference through the identity at initialization.
4. code **Patch size.** ▣ Train DiT-M with patches of size 2 on the pixels for 20,000 updates.
   1. How many tokens does it see, and how many FLOPs per step does `train_step_gflops` count? Is the increase over patch 4 what the cost of attention predicts?
   2. Where does its FD fall on the curve of DiT-S, DiT-M and DiT-L? Does FD depend on the compute per step alone, whatever produces it?
5. code **Large patches and the clean-image target.** ▣ Train `DiT` on pixels with `patch_size=7`, so that each of its 16 tokens carries 49 pixel values, at width 32 with one head and at width 256, for 10,000 updates each. Give each width two output parameterizations under the same velocity loss: the network outputs the velocity, or it outputs $\hat{\mathbf{x}}_0$ and the velocity is formed as $(\mathbf{x}_t - \hat{\mathbf{x}}_0)/t$, with t clamped below at 0.05.
   1. Report the FD of the four models in a 2×2 table.
   2. Does the clean-image output help at width 32, where a token carries more values than the network has channels, and does the gap close at width 256?
   3. Explain the result with the argument of Section 17.5: an image lies near a set of much lower dimension than its pixel count, while the noise fills every dimension. Compare with the finding of Li and He (2026) .
6. code **Width of the bottleneck.** Train autoencoders with 2, 4 and 8 latent channels.
   1. Report the reconstruction FD of each and the decoded FD of a latent U-Net trained on each.
   2. Which of the two numbers moves more, and does the gap between them close as the latent grows? What does this suggest about the four channels of Stable Diffusion's autoencoder, and the 16 or more of later ones?
7. code **The KL weight.** Train the autoencoder with $\beta \in \{0, 10^{-3}, 10^{-2}\}$.
   1. For each, print the latent standard deviation, the encoder's standard deviation, the excess kurtosis and the reconstruction FD.
   2. Train a latent U-Net on each and report the decoded FD. Which β gives the best samples, and why is it not the one whose latents are closest to normal?
   3. With $\beta = 0$ nothing ties the scale of the latents to one. Why does the scale factor c still make the latents usable?
8. **Captions and guidance.** A text-to-image model is trained with the caption replaced

by the empty caption 10% of the time.

1. What distribution does the network learn to sample when it is given the empty caption? What would guidance do if the caption were never dropped?
2. Guidance needs two predictions per step. Compare the cost of two forward passes with that of one pass on a doubled batch, for a DiT with cross-attention and for an MMDiT, where the empty caption also changes the length of the joint sequence.

9. **Cross-attention as a lookup.** Image tokens $\mathbf{Y} \in \mathbb{R}^{m \times d}$ query caption tokens $\mathbf{X} \in \mathbb{R}^{n \times d}$, with queries $\mathbf{Y}\mathbf{W}_q$, keys $\mathbf{X}\mathbf{W}_k$ and values $\mathbf{X}\mathbf{W}_v$.
   1. Show that each image position receives a convex combination of the caption's value vectors. With an empty caption of one token, show that every position receives the same vector, so that the unconditional prediction of guidance sees cross-attention as a learned constant shift.
   2. Suppose that the values were computed from the image instead, $\mathbf{V} = \mathbf{Y}\mathbf{W}_v$. What goes wrong with the shapes when $m \neq n$, and what would such a layer compute when $m = n$?
   3. In cross-attention the caption's keys and values do not depend on $\mathbf{x}_t$; in an MMDiT the caption tokens are updated by the image tokens in every block. For a caption of 77 tokens, 1,024 image tokens and 50 steps with guidance, which caption-side computations can a cross-attention model do once per sample, and what fraction of an MMDiT's token computations belongs to the caption?

   Part (b) is adapted from CMU 10-423/623, Generative AI, Homework 4, Question 2.3 (Spring 2026).
10. code **Scaling in the latent space.** ▣ Train DiT-S and DiT-L on the latents, as DiT-M was trained above.
    1. Add the three latent models to the plot of FD against FLOPs per step, with the reconstruction FD as a horizontal line. Does the latent curve keep falling with size?
    2. Explain the shape of the latent curve through the decoder. What would you change to move it down?
11. code **Compute per step and compute in total.** ▣ Change the training of the three pixel DiTs so that each is scored after 5,000, 10,000 and 20,000 updates.
    1. Plot FD against total training compute, the FLOPs per step times the number of updates, with one curve per model.
    2. At equal total compute, does the larger model reach the lower FD? Compare with the plot of FD against total training compute in the paper of Peebles and Xie, and say what longer runs could change.
    3. At 10,000 updates, does DiT-L still beat the U-Net?
12. [extended] **Latent diffusion for sprites.** ▣▣ Adapt this section's autoencoder to the 64×64 color sprites of Section 16.5, which gives a $16 \times 16 \times 4$ latent, and train a latent U-Net on it. Score the samples with the `d2l.FeatureCNN` of Section 16.5 and compare them with the GANs trained there. Which of the two methods needs more compute for the same FD?

17.7 Few-Step Generation

Every sample from the models of this chapter costs many evaluations of a network. Section 17.4 brought the thousand steps of the ancestral chain down to about fifty, and Section 17.6 made each evaluation cheaper; the generator of a GAN in Chapter 16 needed one.

The obvious shortcut, a single Euler step of the probability-flow ODE, fails even with an exact velocity field: every sample lands on the mean of the data, the average that Section 16.7 met in one-step students and escaped by adding a critic.

This section asks what a network must learn instead so that one or two evaluations suffice without a critic. The answer is to learn the solution of the ODE instead of its right-hand side. A *flow map* takes a point at one time directly to where its trajectory is at an earlier time. Song et al. (2023) learned the jump to the data from the trajectories of a trained diffusion model and called the result a consistency model. Geng et al. (2025) learned the average velocity over any interval with no trained model at all, from an identity that the flow map satisfies.

A flow map keeps the bijection between noise and samples, so inversion and editing work as in Section 17.4, and training remains a regression with one network. We look at the failure on the running example, derive one loss for consistency models and MeanFlow, test it on the mixture and on Fashion-MNIST, and end with the route that matches distributions instead.

```
torch.set_float32_matmul_precision('high')  # tf32 matmuls on tensor cores
```

17.7.1 Why One Step Fails

A long Euler step makes two errors. It aims where the trajectory points at its start, and it ignores how the trajectory bends on the way. Both can be seen on the mixture with the exact field, where no network error hides them.

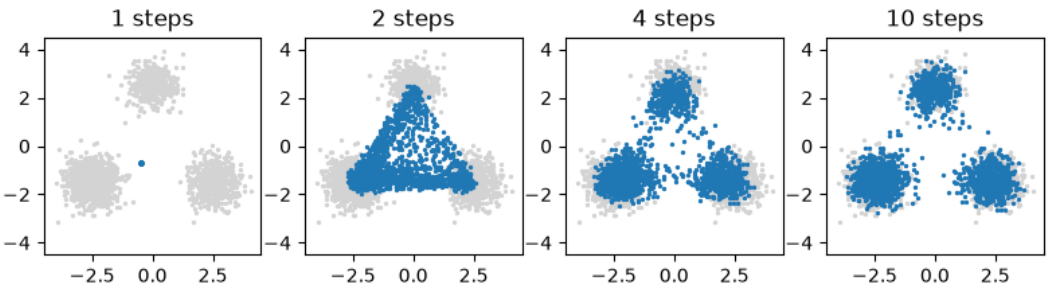
17.7.1.1 The Average Where the Data Are Not

One Euler step from pure noise sends every start to the data mean (Section 17.2), for the mixture $(-0.5, -0.7)$, where no data lie.

More steps help, but how many are needed? `mix.velocity` evaluates the exact field of the mixture, and `d2l.make_pred` turns it into the predictor that `d2l.ode_sample` integrates. We run Euler's method with 1, 2, 4 and 10 steps from the same 20,000 Gaussian starts.

```
device = d2l.try_gpu()
mix = d2l.GaussianMixture.three_modes(device)
lin = d2l.GaussianPath.linear()
pred_exact = d2l.make_pred(mix.velocity, lin, 'velocity')
torch.manual_seed(0)
z = torch.randn(20_000, 2, device=device)
euler = {n: d2l.ode_sample(pred_exact, z, lin.sigma_bars(n))
         for n in (1, 2, 4, 10)}
for n, x in euler.items():
    s = mix.summary(x)
    print(f'{n:2d} Euler steps: weights {s["weights"]}, stray {s["stray"]:.1%}')
d2l.scatter_panels([x[:2000] for x in euler.values()],
                   [f'{n} steps' for n in euler], data=mix.sample(2000),
                   size=2.4);
```

```
 1 Euler steps: weights [1.0, 0.0, 0.0], stray 100.0%
 2 Euler steps: weights [0.644, 0.217, 0.14], stray 25.1%
 4 Euler steps: weights [0.54, 0.277, 0.183], stray 6.5%
10 Euler steps: weights [0.512, 0.291, 0.197], stray 2.2%
```



One step puts every sample on the mean, farther than three standard deviations from every mode, so all of them stray. Two steps reach the modes but leave about a quarter of the samples between them, with weights of about 0.64/0.22/0.14 instead of 0.5/0.3/0.2: the heaviest mode gains because the mean lies nearest to it. Four steps still leave about 6.5% of the samples stray, and ten steps about 2%.

17.7.1.2 Straighter Paths Help but Do Not Finish

Why do the trajectories bend at all? Each training example joins a data point to an independently drawn noise vector by a straight segment, but the segments of different examples cross. Where they cross, the field averages their directions, and the trajectories, which cannot cross, bend around each other (Section 38.6.2). The curvature comes from the pairing of noise with data, not from the loss.

Reflow (Liu et al. 2023) changes the pairing. It pairs each noise vector with the sample that a trained model reaches from it and trains the network again on these pairs, whose segments rarely cross. Section 38.6.2.3 proves that a round of reflow keeps the data distribution and never increases the transport cost, and on two-moons data one Euler step of the reflowed model scores as well as thirty-two steps of the original.

Straightening does not finish the job. The reflowed model inherits the original's errors, each round costs a training run, and on images Liu et al. (2023) still followed reflow with distillation for their one-step results. The rest of this section changes what the network outputs instead of how the data are paired.

17.7.2 Flow Maps

A velocity model says which way a trajectory points; a flow map says where it arrives.

17.7.2.1 The Integral of the Velocity

Run the probability-flow ODE $d\mathbf{x}_\tau/d\tau = \mathbf{u}_\tau(\mathbf{x}_\tau)$ from a point $\mathbf{x}$ at time t back to an earlier time $s \le t$. The point where it arrives is the *flow map* $\Phi_{t\to s}(\mathbf{x})$. For a field that is Lipschitz in $\mathbf{x}$ the flow map is a bijection (Section 38.1.1). The sampler we want is $\Phi_{1\to 0}$.

Integrating the ODE along the trajectory $\mathbf{x}_\tau$ through $\mathbf{x}_t$ writes the flow map as a displacement,

$$\Phi_{t\to s}(\mathbf{x}_t) = \mathbf{x}_t - \int_s^t \mathbf{u}_\tau(\mathbf{x}_\tau)\, d\tau = \mathbf{x}_t - (t-s)\,\bar{\mathbf{u}}(\mathbf{x}_t, s, t), \tag{17.54}$$

where $\bar{\mathbf{u}}(\mathbf{x}_t, s, t)$ is the *average velocity* along the trajectory between the two times. Euler's method replaces it by the velocity at the start, $\mathbf{u}_t(\mathbf{x}_t)$, with an error that grows with the step (Section 38.1.3). Over an interval of length zero the two agree, $\bar{\mathbf{u}}(\mathbf{x}, t, t) = \mathbf{u}_t(\mathbf{x})$, so a network for $\bar{\mathbf{u}}$ is a velocity model with one more input, the end time.

The smallest case shows what the average velocity knows that the velocity does not. Let the data be one-dimensional and Gaussian with mean m and standard deviation ς. Then x_t is Gaussian with mean $(1-t)\,m$ and standard deviation $k_t = \sqrt{(1-t)^2\varsigma^2 + t^2}$, and the flow keeps each point's standardized position $(x - (1-t)m)/k_t$, so the flow map is affine:

$$\Phi_{t\to s}(x) = (1-s)\,m + \frac{k_s}{k_t}(x - (1-t)\,m).$$

With $m = 2$ and $\varsigma = 0.5$ the full jump $\Phi_{1\to 0}(x) = 2 + 0.5\,x$ sends standard Gaussian noise to the data $\mathcal{N}(2, 0.5^2)$, with average velocity $\bar{u}(x, 0, 1) = 0.5\,x - 2$. The velocity at the start is $u_1(x) = x - 2$, and its Euler step returns 2 for every x. The average velocity removes only half of the noise, because half of the noise's spread must survive as the spread of the data.

Three identities characterize the flow map, and each gives a way to train one (Figure 17.14). Let $\mathbf{x}_\tau$ be the trajectory through $\mathbf{x}$ at time t, and $s \le r \le t$. The *compositional* identity says that two jumps make one, $\Phi_{r\to s}(\Phi_{t\to r}(\mathbf{x})) = \Phi_{t\to s}(\mathbf{x})$. The *Lagrangian* identity fixes the start and moves the end: the mapped point moves with the field, like a particle, $\partial_s \Phi_{t\to s}(\mathbf{x}) = \mathbf{u}_s(\Phi_{t\to s}(\mathbf{x}))$. The *Eulerian* identity fixes the end and moves the start along the trajectory: every point of one trajectory maps to the same place,

$$\frac{d}{dt}\Phi_{t\to s}(\mathbf{x}_t) = \partial_t \Phi_{t\to s}(\mathbf{x}_t) + \nabla_{\mathbf{x}} \Phi_{t\to s}(\mathbf{x}_t)\,\mathbf{u}_t(\mathbf{x}_t) = \mathbf{0}. \tag{17.55}$$

In the Gaussian example all three can be checked by hand: the ratios k_s/k_t compose, and the standardized position is constant along a trajectory.

Together with the condition at $s = t$, where the map is the identity and its slope in s is the velocity, each identity determines the flow map. Composition needs only the model at an intermediate time; progressive distillation (Salimans and Ho 2022) and shortcut models (Frans et al. 2025) use it. The Lagrangian identity needs the velocity at the output of the map. The Eulerian identity needs the derivative along the trajectory through the input, one Jacobian-vector product. Boffi et al. (2025) sort the few-step methods by the identity they enforce; both methods of this section are Eulerian.

17.7.2.2 Consistency Models

The first flow-map method kept only the jump to the data. A *consistency model* (Song et al. 2023) is a network $f(\mathbf{x}, t)$ trained to output $\Phi_{t\to 0}(\mathbf{x})$ at every time. Its training condition is that the output does not change as the input moves along a trajectory, $\frac{d}{dt} f(\mathbf{x}_t, t) = \mathbf{0}$ with $f(\mathbf{x}, 0) = \mathbf{x}$, the Eulerian identity Equation 17.55 at $s = 0$.

In *consistency distillation*, Song et al. let a trained diffusion model, the *teacher*, take one ODE step between neighbouring times of a grid, and require the student's outputs at the two points to agree, with the output at the earlier point held fixed as the target. As the grid is refined the difference, divided by the step, becomes the derivative above, which Lu and Song

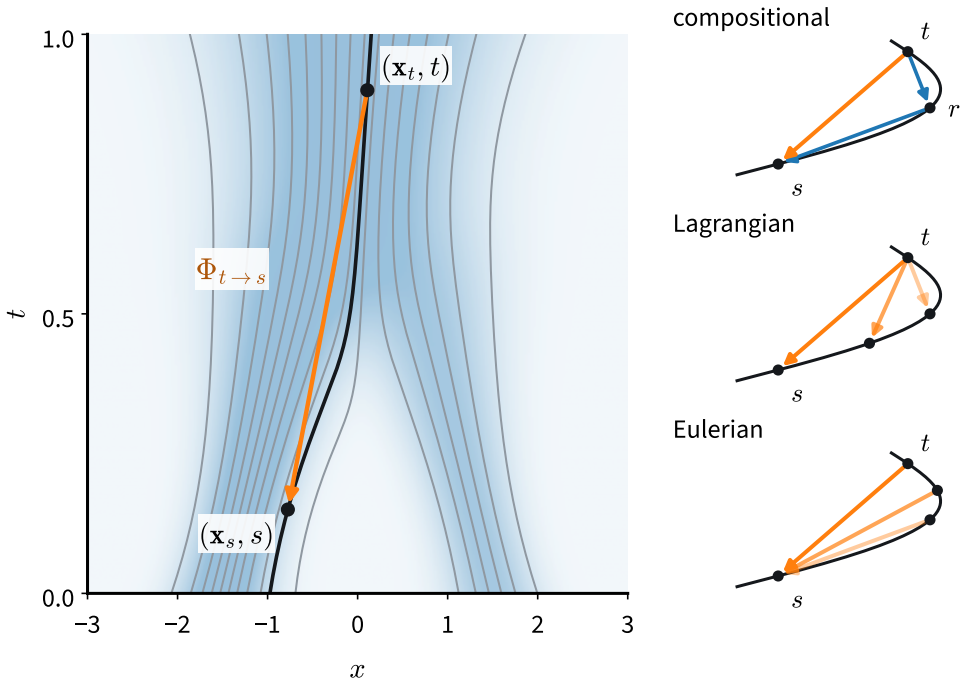


Figure 17.14: The flow map. Left: trajectories of the probability-flow ODE for a one-dimensional mixture of two Gaussians, from the noise at $t = 1$ (top) to the data at $t = 0$ (bottom); the chord from $(\mathbf{x}_t, t)$ to $(\mathbf{x}_s, s)$ is one jump of the flow map. Right: the compositional, Lagrangian and Eulerian identities.

(2025) use directly. *Consistency training* replaces the teacher's velocity by the conditional velocity of each example.

One evaluation of a consistency model gives a sample. A second cannot jump from $t = 1$ to $t = 0.5$, because the model knows only where trajectories end. It *re-noises* instead: the sample is moved back to $t = 0.5$ with fresh noise, $\mathbf{x}_{0.5} = 0.5\,\mathbf{x}_0 + 0.5\,\epsilon'$, and mapped to the data again, from where trajectories bend less. A flow map with both times needs no re-noising: two evaluations are two jumps along one trajectory, and the bijection between noise and samples survives.

17.7.2.3 Average Velocity and the MeanFlow Identity

Writing the flow map through the average velocity Equation 17.54 turns the Eulerian identity into a regression target. Multiply the average velocity by the length of its interval, which gives the integral of the velocity, and differentiate in t along the trajectory with s held fixed:

$$\frac{d}{dt}\Big[(t-s)\,\bar{\mathbf{u}}(\mathbf{x}_t, s, t)\Big] = \frac{d}{dt}\int_s^t \mathbf{u}_\tau(\mathbf{x}_\tau)\,d\tau = \mathbf{u}_t(\mathbf{x}_t).$$

The product rule on the left gives the *MeanFlow identity* of Geng et al. (2025),

$$\bar{\mathbf{u}}(\mathbf{x}_t, s, t) = \mathbf{u}_t(\mathbf{x}_t) - (t-s)\,\frac{d}{dt}\bar{\mathbf{u}}(\mathbf{x}_t, s, t). \tag{17.56}$$

The average velocity is the velocity at the start, corrected by how fast the average changes as the start slides along the trajectory. Substituting Equation 17.54 into Equation 17.55 shows that the two identities are the same. The total derivative is $\partial_t \bar{\mathbf{u}} + (\nabla_{\mathbf{x}} \bar{\mathbf{u}})\,\mathbf{u}_t$, the Jacobian-vector product of the network $(\mathbf{x}, t) \mapsto \bar{\mathbf{u}}(\mathbf{x}, s, t)$ with the tangent $(\mathbf{u}_t, 1)$, which forward-mode differentiation computes together with the output at about the cost of a backward pass.

The identity becomes a loss by regressing the network onto its right-hand side, computed from the network itself and held fixed. The marginal velocity $\mathbf{u}_t$ is not available, but two substitutes are. A teacher provides its estimate $\hat{\mathbf{u}}$ of $\mathbf{u}_t(\mathbf{x}_t)$. Without one, the conditional velocity $\epsilon - \mathbf{x}_0$ of the training example serves, for the reason that made flow matching work. The target is affine in the velocity, directly and through the tangent of the Jacobian-vector product. Averaged over the examples that could have produced $\mathbf{x}_t$, it is therefore the target with $\mathbf{u}_t(\mathbf{x}_t)$ in place of the conditional velocity, and squared error is minimized by that average. Writing $\mathbf{u}_{\text{ref}}$ for the velocity we use, the *reference velocity*, the loss at a training point is

$$\Big\|\bar{\mathbf{u}}_\theta(\mathbf{x}_t, s, t) - \text{sg}\Big[\mathbf{u}_{\text{ref}} - (t-s)\,\frac{d}{dt}\bar{\mathbf{u}}_\theta(\mathbf{x}_t, s, t)\Big]\Big\|^2, \tag{17.57}$$

where sg, the *stop-gradient*, marks a quantity treated as a constant when the loss is differentiated. It is required. Without it the target moves with every update of the network, and training on the mixture fails: almost every sample strays (Exercise 6).

The loss leaves two choices, where $\mathbf{u}_{\text{ref}}$ comes from and whether the end time s is free or fixed at zero, and the methods in the literature are the four combinations.

Table 17.2: Diffusion Flow Map Methods

velocity $\mathbf{u}_{\text{ref}}$	one time, $s = 0$	two times, $s \le t$
a teacher's $\hat{\mathbf{u}}$	consistency distillation	flow-map distillation (Kim et al. 2024)
the example's $\epsilon - \mathbf{x}_0$	consistency training	MeanFlow

At $s = 0$, Equation 17.54 reads $f(\mathbf{x}, t) = \mathbf{x} - t\,\bar{\mathbf{u}}(\mathbf{x}, 0, t)$, so the top-left cell is consistency distillation in the continuous-time form of Lu and Song (2025) . `flow_map_error` implements Equation 17.57 per example on the linear path for all four: the network `u` computes $\bar{\mathbf{u}}$, `teacher=None` selects the conditional velocity, and the caller chooses the times and averages.

```
def flow_map_error(u, teacher, x0, eps, s, t):
    """Per-example squared error of the average velocity u(x_t, s, t) against
    the stop-gradient target v - (t - s) du/dt; v from teacher, or eps - x0."""
    x_t = d2l.GaussianPath.linear().sample(x0, t, eps)
    if teacher is None:
        v = eps - x0
    else:
        with torch.no_grad():
            v = teacher(x_t, t)
    u_t, du_dt = torch.func.jvp(lambda x, r: u(x, s, r), (x_t, t),
                                (v, torch.ones_like(t)))
    gap = (t - s).reshape(-1, *[1] * (x0.dim() - 1))
    target = (v - gap * du_dt).detach()
    return ((u_t - target) ** 2).flatten(1).sum(1)
```

The derivative in t passes through the network's time features, which must vary slowly. `d2l.time_embedding` uses sines and cosines of t at frequencies up to one. Diffusion code often embeds $1000\,t$, which puts frequencies up to a thousand into $d\bar{\mathbf{u}}/dt$ and turns the target into noise, a source of instability that Lu and Song (2025) report for continuous-time consistency models.

A flow map takes $\mathbf{x}_t$ to $\mathbf{x}_s$ in one jump, $\mathbf{x}_s = \mathbf{x}_t - (t - s)\,\bar{\mathbf{u}}(\mathbf{x}_t, s, t)$, so a few-step sampler cuts $[0, 1]$ into equal intervals and jumps across each.

```
@torch.inference_mode()
def flow_map_sample(u, x, steps, t_start=1.0):
    """Jump from x at t_start (1 is noise) to data at t = 0 in equal steps."""
    ts = torch.linspace(t_start, 0, steps + 1, device=x.device)
    for t, s in zip(ts[:-1], ts[1:]):
        x = x - (t - s) * u(x, s.expand(len(x)), t.expand(len(x)))
    return x
```

With the velocity $\mathbf{u}_t$ in place of $\bar{\mathbf{u}}$, the same function is Euler's method, so one sampler serves the teacher and the student.

On images the student starts as a copy of the teacher's U-Net. The gap $t - s$ enters through its own time embedding and a linear map initialized at zero, added to the U-Net's embedding, so that at the start $\bar{\mathbf{u}}(\mathbf{x}, s, t) = \hat{\mathbf{u}}(\mathbf{x}, t)$, the teacher's velocity.

```
class UNetFlowMap(nn.Module):
    """u(x, s, t): a copy of a velocity U-Net, plus a zero-initialized gap."""
    def __init__(self, unet):
        super().__init__()
        import copy  # the d2l library preamble does not import it
        # the teacher is an EMA copy with gradients off; the student trains
        self.unet = copy.deepcopy(unet).requires_grad_(True)
        self.gap = nn.Linear(unet.emb_dim, unet.emb_dim)
        nn.init.zeros_(self.gap.weight), nn.init.zeros_(self.gap.bias)

    def forward(self, x, s, t):
        gap = self.gap(d2l.time_embedding(t - s, self.unet.emb_dim))
        return self.unet(x, t, cond=gap)
```

17.7.3 Experiments

17.7.3.1 On the Running Example

The teacher is the velocity model `TimeMLP` of Section 17.2, trained here for 20,000 steps.

```
torch.manual_seed(1)
_, teacher, _ = d2l.train_regression(
    d2l.TimeMLP(256, 3).to(device), d2l.regression_loss(lin, 'velocity'),
    lambda: mix.sample(512), 20_000, cosine=True)
```

Both students are `TimeMLP`s that see the start t and the interval $t - s$. The consistency student draws t uniformly, fixes $s = 0$ and uses the teacher's velocity. The MeanFlow student draws two uniform times, takes the smaller as s, and uses no teacher. Everything else is the teacher's recipe.

```
class MLPFlowMap(d2l.TimeMLP):
    """u(x, s, t): a TimeMLP on the start t and the interval t - s."""
    def forward(self, x, s, t):
        return super().forward(x, t, t - s)

def consistency_loss(u, x0):  # s = 0, the teacher's velocity
    t = torch.rand(len(x0), device=device)
    return d2l.flow_map_error(u, teacher, x0, torch.randn_like(x0),
                              torch.zeros_like(t), t).mean()

def meanflow_loss(u, x0):  # s < t, no teacher
    a, b = torch.rand(2, len(x0), device=device)
    return d2l.flow_map_error(u, None, x0, torch.randn_like(x0),
                              torch.minimum(a, b), torch.maximum(a, b)).mean()

students = {}
for name, loss in (('consistency', consistency_loss),
                   ('MeanFlow', meanflow_loss)):
    torch.manual_seed(2)
    _, students[name], _ = d2l.train_regression(
        MLPFlowMap(256, 3, num_times=2).to(device), loss,
```

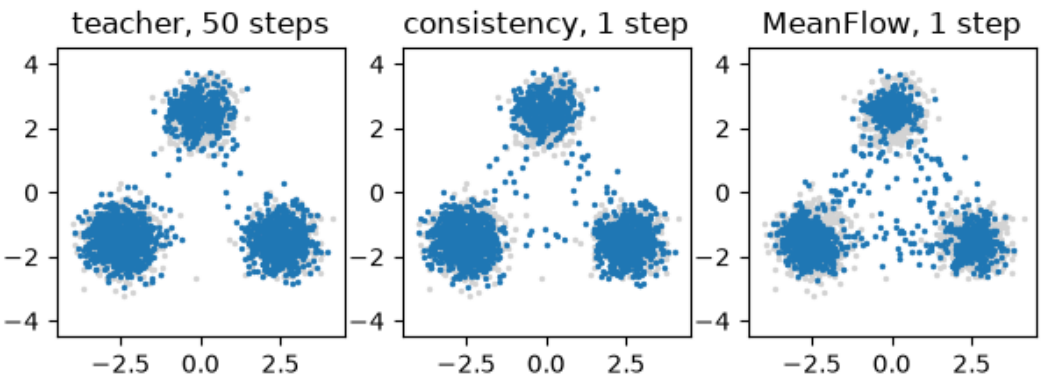
```
        lambda: mix.sample(512), 20_000, cosine=True)
```

All models start from the 20,000 points of the Euler experiment. The teacher takes fifty Euler steps, the consistency student one jump or two with re-noising at $t = 0.5$, and the MeanFlow student one, two or four jumps. A fresh draw from the mixture shows what a perfect sampler would print.

```
def renoised_sample(u, z):
    """Two evaluations of a one-time map: jump, re-noise to t = 0.5, jump."""
    x = d2l.flow_map_sample(u, z, 1)
    half = torch.full((len(z),), 0.5, device=device)
    return d2l.flow_map_sample(u, lin.sample(x, half, torch.randn_like(x)), 1,
                               t_start=0.5)

cm, mf = students['consistency'], students['MeanFlow']
samples = {'data': mix.sample(20_000),
           'teacher, 50 steps': d2l.flow_map_sample(
               lambda x, s, t: teacher(x, t), z, 50),
           'consistency, 1': d2l.flow_map_sample(cm, z, 1),
           'consistency, 2 re-noised': renoised_sample(cm, z),
           **{f'MeanFlow, {k}': d2l.flow_map_sample(mf, z, k)
              for k in (1, 2, 4)}}
for name, x in samples.items():
    s = mix.summary(x)
    print(f'{name:>24}: weights {s["weights"]}, stray {s["stray"]:.1%}, '
          f'spread {s["spread"]:.3f}')
d2l.scatter_panels([samples[k][:2000] for k in
                    ('teacher, 50 steps', 'consistency, 1', 'MeanFlow, 1')],
                   ['teacher, 50 steps', 'consistency, 1 step',
                    'MeanFlow, 1 step'], data=mix.sample(2000), size=2.4);
```

```
                    data: weights [0.498, 0.302, 0.199], stray 1.2%, spread 0.476
       teacher, 50 steps: weights [0.501, 0.297, 0.202], stray 1.7%, spread 0.459
          consistency, 1: weights [0.498, 0.298, 0.204], stray 2.2%, spread 0.454
consistency, 2 re-noised: weights [0.498, 0.3, 0.202], stray 1.3%, spread 0.474
             MeanFlow, 1: weights [0.507, 0.29, 0.203], stray 4.2%, spread 0.313
             MeanFlow, 2: weights [0.496, 0.305, 0.199], stray 1.5%, spread 0.361
             MeanFlow, 4: weights [0.494, 0.308, 0.198], stray 0.8%, spread 0.368
```



Both students reproduce the mixture in one evaluation, with weights within about 0.015 of 0.5/0.3/0.2. The fresh draw strays about 1.1% of the time, the tail of each Gaussian beyond three standard deviations, and the teacher with fifty Euler steps about 1.5%. The consistency student strays about 2% after one evaluation. The MeanFlow student, trained without a teacher, strays about 4%, and its other samples sit closer to the mode centers than the data do. A second evaluation, re-noised or a second jump, brings the strays of both to the teacher's level. MeanFlow's samples stay too tight, though: their spread is about 0.38 after two and four jumps against 0.47 for the data, which is also why four jumps stray less often than the data themselves.

MeanFlow's ImageNet recipe divides each example's squared error e by $(e+c)^p$, held fixed, with $p = 1$ and a small c, so that the gradient is that of $\log(e+c)$. Squared error is minimized by the conditional mean of the target, its logarithm near the most probable value. Without a teacher the targets at a noisy point spread over the three modes, so the weighted loss should pull the field toward the heaviest mode. We train the MeanFlow student again with this weight.

```
def weighted_meanflow_loss(u, x0, c=1e-3):
    """The MeanFlow loss with each squared error e divided by e + c, fixed."""
    a, b = torch.rand(2, len(x0), device=device)
    e = d2l.flow_map_error(u, None, x0, torch.randn_like(x0),
                           torch.minimum(a, b), torch.maximum(a, b))
    return (e / (e.detach() + c)).mean()

torch.manual_seed(2)
_, weighted, _ = d2l.train_regression(
    MLPFlowMap(256, 3, num_times=2).to(device), weighted_meanflow_loss,
    lambda: mix.sample(512), 20_000, cosine=True)
s = mix.summary(d2l.flow_map_sample(weighted, z, 1))
print(f'weighted MeanFlow, 1: weights {s["weights"]}, stray {s["stray"]:.1%}')
```

```
weighted MeanFlow, 1: weights [0.944, 0.027, 0.029], stray 2.3%
```

The weighted student puts more than nine samples in ten on the heaviest mode, and few of them stray: the loss has fitted the most probable target instead of the average one. Plain squared error recovers the weights, which is why the other losses take the plain mean of `flow_map_error`.

17.7.3.2 On Images

On Fashion-MNIST we distil the unconditional U-Net of Section 17.4, about 1.2 million parameters trained for 10,000 updates, through the moving average of its weights. We train it again here.

```
X, y, X_test, y_test = d2l.load_fashion_mnist_pm1()

def batch_fn():
    return X[torch.randint(0, len(X), (128,), device=X.device)]

torch.manual_seed(0)
_, teacher, _ = d2l.train_regression(d2l.UNet(ch=64).to(X.device),
```

```
                                    d2l.regression_loss(lin, 'velocity'),
                                    batch_fn, 10_000)
```

We score 1,000 samples by the Fréchet distance (FD) of Section 17.3 against 1,000 test images, read against the floor of two disjoint sets of 1,000 test images.

```
cnn = d2l.fashion_feature_cnn(X, y)
features = lambda imgs: d2l.fd_features(cnn, imgs)
f_test = features(X_test[:1000])
floor = d2l.fid_score(features(X_test[1000:2000]), f_test)
print(f'floor, 1,000 real images against 1,000 others: FD {floor:.2f}')
```

```
floor, 1,000 real images against 1,000 others: FD 0.81
```

The student is `UNetFlowMap` on the teacher's averaged weights, trained with the teacher's velocity and both times free. Half of the pairs have $s = t$, where the loss is the teacher's own velocity loss; the other half draw $s < t$ as on the mixture. Starting from trained weights, it learns at a rate of $2 \cdot 10^{-4}$ for 5,000 updates, half the teacher's 10,000.

```
def student_loss(u, x0):
    a, b = torch.rand(2, len(x0), device=x0.device)
    s, t = torch.minimum(a, b), torch.maximum(a, b)
    # half the pairs have s = t: the plain flow-matching term
    s = torch.where(torch.rand(len(x0), device=x0.device) < 0.5, t, s)
    return d2l.flow_map_error(u, teacher, x0, torch.randn_like(x0), s,
                              t).mean()

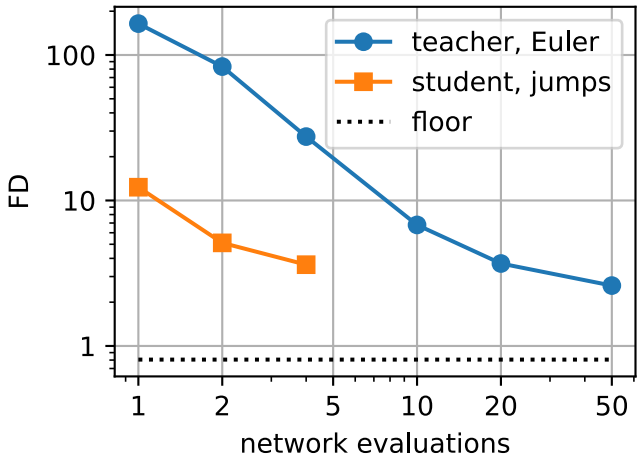
torch.manual_seed(2)
_, student, _ = d2l.train_regression(  # jvp through GroupNorm won't compile
    d2l.UNetFlowMap(teacher).to(X.device), student_loss, batch_fn, 5_000,
    lr=2e-4, ema_half_life=500, compiled=False)
```

From the same 1,000 noise images, the teacher takes 1 to 50 Euler steps and the student 1, 2 or 4 jumps, and we plot FD against the number of network evaluations.

```
torch.manual_seed(3)
noise = torch.randn(1000, 1, 28, 28, device=X.device)
velocity = lambda x, s, t: teacher(x, t)
ns_teacher, ns_student = (1, 2, 4, 10, 20, 50), (1, 2, 4)
images = {('teacher', n): d2l.flow_map_sample(velocity, noise, n)
          for n in ns_teacher}
images |= {('student', n): d2l.flow_map_sample(student, noise, n)
           for n in ns_student}
fd = {key: d2l.fid_score(features(x), f_test) for key, x in images.items()}
for (name, n), v in fd.items():
    print(f'{name}, {n:2d} evaluations: FD {v:6.2f}')
d2l.plot([ns_teacher, ns_student, [1, 50]],
         [[fd['teacher', n] for n in ns_teacher],
          [fd['student', n] for n in ns_student], [floor, floor]],
         'network evaluations', 'FD',
```

```
            legend=['teacher, Euler', 'student, jumps', 'floor'],
            xscale='log', yscale='log', fmts=('o-', 's-', 'k:'))
```

```
teacher,  1 evaluations: FD 164.34
teacher,  2 evaluations: FD  83.23
teacher,  4 evaluations: FD  27.49
teacher, 10 evaluations: FD   6.79
teacher, 20 evaluations: FD   3.69
teacher, 50 evaluations: FD   2.60
student,  1 evaluations: FD  12.34
student,  2 evaluations: FD   5.11
student,  4 evaluations: FD   3.62
```

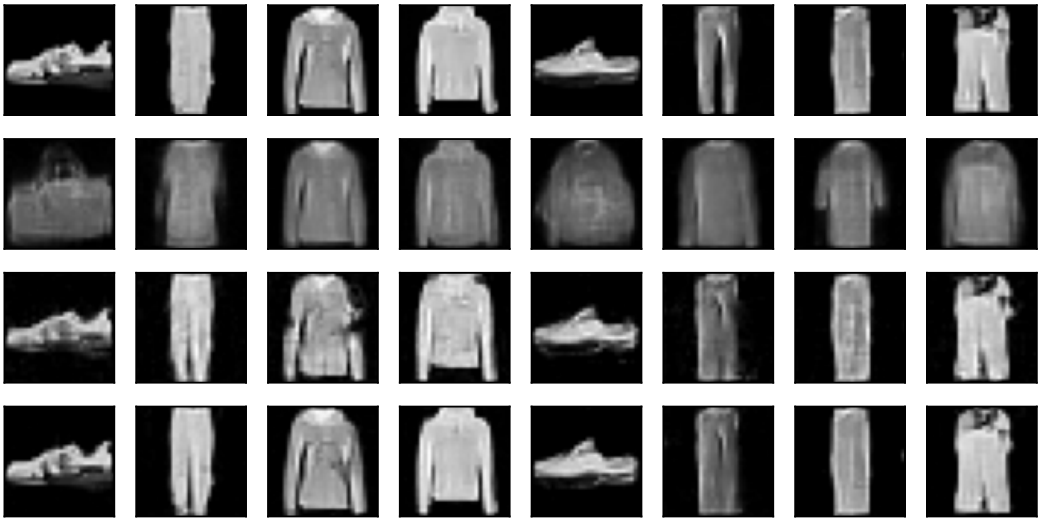


The teacher's FD falls from about 150 at one Euler step and 70 to 85 at two to about 7 at ten and 2.5 to 2.8 at fifty, against a floor of about 0.8. The student lies far below this curve at small budgets: one jump scores about 4 to 5 times the teacher's fifty-step FD, two jumps about twice, and four within about 1.5 times. Two evaluations of the student are worth about fifteen Euler steps of the teacher, and one about seven.

The remaining gap at one evaluation is a matter of budget. The students in the papers train for roughly 100,000 to 800,000 iterations at batches of 512, two to three orders of magnitude more than ours, and report one-step FIDs about 1.4 to 1.7 times those of their teachers. Training our student longer lowers its one-step FD only slowly.

The rows below are drawn from the same eight noise images: the teacher with fifty and with two Euler steps, and the student with one and with two jumps.

```
rows = [('teacher', 50), ('teacher', 2), ('student', 1), ('student', 2)]
d2l.show_gray(torch.cat([images[r][:8].clamp(-1, 1) for r in rows]), 4, 8);
```



The rows match column by column. Two Euler steps of the teacher draw grey averages of garments, the collapse of the first section in pixels. The student draws the teacher's garment from each noise image, sharp but over-contrasted and sometimes broken after one jump and close to the teacher's after two.

17.7.4 The Other Route: Matching Distributions

A flow map regresses onto a target computed from an ODE's trajectories, so its student can be no better than those trajectories. The distillation methods of Section 16.7 instead train a one-step generator to match the distribution of the data, with a critic against real images (Sauer, Lorenz, et al. 2023) or with the difference between the teacher's score and that of the student's own samples (Yin, Gharbi, Zhang, et al. 2024). Both give up the bijection, so inversion and editing through the ODE no longer apply. They gain quality in one step, and with real images in the loss a student can surpass its teacher (Section 16.8).

Classifier-free guidance (Section 17.3) evaluates the network twice per step. Meng et al. (2023) distil the guided velocity into one network that takes the guidance strength as an input, which halves the cost of every step before the number of steps is reduced (Exercise 8).

17.7.5 Summary

One Euler step from noise lands on the mean of the data, and straightening the field reduces the number of steps but not to one. A flow map learns where a trajectory arrives instead of where it points. Differentiating its definition along the trajectory gives the MeanFlow identity, a regression target that needs one Jacobian-vector product and either a teacher's velocity or each example's conditional velocity; with $s = 0$ and a teacher it is consistency distillation. On the mixture one evaluation recovers the mode weights; on images a short student comes within about 1.5 times the teacher's FD in four evaluations. All of this needs a continuous path and a velocity, which do not exist when $\mathbf{x}$ is a sequence of tokens, the subject of Section 17.8.

17.7.6 Exercises

1. **One and two Euler steps.**
   1. A single step of the deterministic sampler from pure noise to $t = 0$ returns $\mathbb{E}[\mathbf{x}_0]$ from every start, whatever the path (Section 17.2). Does the same hold for one Euler step in t, $\mathbf{x} - \mathbf{u}_1(\mathbf{x})$, on the cosine path $\alpha_t = \cos\frac{\pi t}{2}$, $\sigma_t = \sin\frac{\pi t}{2}$? What does that step return, and why does an Euler step in t depend on the path when a step in $\bar{\sigma}$ does not?
   2. Let the data be the two points $\pm a$ in one dimension, with equal probability, on the linear path. Show that $\mathbb{E}[x_0 \mid x_t = x] = a\tanh\left(a(1-t)x/t^2\right)$ and that two Euler steps of length $\frac{1}{2}$ map a start x to $a\tanh(ax)$. Section 38.6.1.3 has the exact flow of this example.
   3. For $a = 2$, what fraction of the two-step samples lies within 0.1 of a data point? Where does the exact flow map $\Phi_{1\to 0}$ send x?
2. code **Reflow on the mixture.** ▣ Reflow (Liu et al. 2023) retrains a velocity model on pairs of a noise vector and the sample that the current model produces from it.
   1. Draw 20,000 noise vectors, integrate the teacher of this section with fifty Euler steps, and train a new `TimeMLP` on the pairs with the velocity loss and the teacher's settings.
   2. Report the weights and strays of the new model at 1, 2, 4 and 10 Euler steps, and the mean distance of its fifty-step trajectories from their chords, next to the same quantities for the teacher.
   3. Which of the teacher's errors does reflow remove, which does it inherit, and which does it add? What would a second round do to each?
3. code **Progressive distillation.** ▣ Salimans and Ho (2022) train a student so that one of its steps reproduces two steps of the teacher, then make the student the teacher and repeat.
   1. On the mixture, start from 64 Euler steps of the teacher of this section and halve the number of steps six times, training each student for a fifth of the teacher's updates. Which identity of the flow map does each round enforce?
   2. Report the strays after each round, and compare the one-step student with the flow-map students of this section at equal total training steps. Where do the errors of the rounds come from?
4. code **Which time pairs to train on.** The MeanFlow student drew every pair with $s < t$, the image student half of them with $s = t$.
   1. Train the MeanFlow student on the mixture with a share of 0.25, 0.5, 0.75 and 1 of the pairs having $s < t$, and report the strays at one and two jumps. Which share serves one jump best?
   2. Replace the uniform times by a logit-normal distribution of t with mean -0.4 and standard deviation 1, which concentrates the times in the middle of the path. What changes, and why?
   3. Why might pairs with $s = t$ matter more on images, where the student starts from the teacher's weights, than on the mixture, where it starts from random weights?
5. code **The derivative against a difference.** ▣ Consistency distillation as published (Song et al. 2023) uses a grid of $N = 18$ times, one Heun step of the teacher from t_n to t_{n-1}, and a target held fixed at the earlier point.
   1. Write the student as $f(\mathbf{x}, t) = \mathbf{x} - t\,\bar{\mathbf{u}}(\mathbf{x}, 0, t)$ and show that the difference of its outputs at the two points, divided by $t_n - t_{n-1}$, tends to the Eulerian derivative Equation 17.55 at $s = 0$ as N grows.

2. Implement the discrete loss without a Jacobian-vector product, train it on the mixture with the teacher of this section, and compare its one-step strays with those of the continuous student. Try $N = 8$ and $N = 64$. Does a finer grid help, and why?

6. code **The stop-gradient.** Copy `flow_map_error` into a cell and remove the stop-gradient on the target.
   1. Train the MeanFlow student on the mixture with this loss and report its one-step weights and strays. Write out the gradient of the loss with the target differentiated, and identify the term that the stop-gradient removes. What can the network change to lower this loss without satisfying Equation 17.56?
   2. Song et al. compute the target with a moving average of the student's weights instead. Implement this with `d2l.EMA` and compare.
7. **The Lagrangian loss.** ▣ Boffi et al. (2025) train flow maps with the Lagrangian identity.
   1. Substitute Equation 17.54 into $\partial_s \Phi_{t\to s}(\mathbf{x}) = \mathbf{u}_s(\Phi_{t\to s}(\mathbf{x}))$ and derive an identity for $\bar{\mathbf{u}}$ that involves the velocity at the output of the jump.
   2. Why can the conditional velocity $\epsilon - \mathbf{x}_0$ not serve in this identity, so that a velocity model is needed? Show that the flow map's own velocity $\bar{\mathbf{u}}(\cdot, s, s)$ can serve, so that one network trains itself.
   3. code Train such a student on the mixture and compare it with the MeanFlow student in strays and in training cost per step.
8. code **A guided student.** ▣ Train the class-conditional U-Net of Section 17.3 and distil it into a two-time student whose reference velocity $\mathbf{u}_{\text{ref}}$ is the guided velocity $\tilde{\mathbf{u}}$, `d2l.cfg(teacher, c, 2.0)`, for random labels c, with the label as an input of the student.
   1. Compare the FD and the agreement of the predicted class with the requested one for the teacher with fifty guided Euler steps and the student with one, two and four jumps.
   2. How many network evaluations does each sample cost in each case?
9. code **Quality against cost.** Measure the wall-clock time per 1,000 images and the FD of the teacher of this section with 1 to 50 Euler steps, with Heun's method on the crowded grid of Section 17.4, and of the student with 1 to 4 jumps.
   1. Plot FD against time. At which FD do the curves cross?
   2. The student cost extra training. How many images must be generated before that training is repaid at the FD of two jumps?
10. **Matching noised distributions.** ▣ Score-difference distillation (Yin, Gharbi, Zhang, et al. 2024) trains a one-step generator $\mathbf{x} = g_\theta(\mathbf{z})$ along $\mathbb{E}\big[(\nabla \log q_{\theta,t}(\mathbf{x}_t) - \nabla \log p_t(\mathbf{x}_t))^\top \partial \mathbf{x}_t / \partial \theta\big]$, where $\mathbf{x}_t = \alpha_t g_\theta(\mathbf{z}) + \sigma_t \epsilon$ and $q_{\theta,t}$ is the law of $\mathbf{x}_t$: the score of its own noised samples minus the teacher's.
    1. Show that this is the gradient of $D_{\text{KL}}(q_{\theta,t} \,\|\, p_t)$. Why does the term from the dependence of $q_{\theta,t}$ itself on θ vanish?
    2. In one dimension, let the generator be $m + sz$ with $s > 0$ and the data $\mathcal{N}(\mu, v)$. Write $D_{\text{KL}}(q_t \,\|\, p_t)$ on the linear path in closed form, and show that $m = \mu$, $s^2 = v$ is its only minimizer at every $t < 1$.
    3. Let the data be $\frac{1}{2}\mathcal{N}(-2, 0.25) + \frac{1}{2}\mathcal{N}(2, 0.25)$. Evaluate $D_{\text{KL}}(q_t \,\|\, p_t)$ by quadrature on a grid of (m, s) at $t \in \{0.1, 0.3, 0.5, 0.9\}$, and locate the minimizer at each t. Why is the minimum near $\log 2$ at small t? Relate the result to mode dropping in GANs (Section 16.1.3.3) and to the merging of modes under noise (6). What does a weighting of the levels toward large t buy a Gaussian generator, and what can no weighting buy it?
11. [extended] **Shortcut models.** ▣▣ A shortcut model (Frans et al. 2025) conditions a ve-

locity model on the step size d and trains it from scratch with two losses: flow matching at $d = 0$, and self-consistency, which asks one step of size $2d$ to equal two steps of size d computed by the model itself and held fixed.

1. Which identity of the flow map does self-consistency enforce, and why does it need no Jacobian-vector product?
2. Implement it on the mixture with step sizes 2^{-k}, $k = 0, \dots, 6$, and compare its strays at one, two and four steps with those of the MeanFlow student at equal training steps.

17.8 Discrete Diffusion

Every model in this chapter has added Gaussian noise to real numbers: coordinates of points, pixel intensities, latents of an autoencoder. The language models of Section 11.2 work with tokens, write text one token at a time from left to right, and never revise a token once it is written. Adding a small real number to the index of the word "cat" does not produce a slightly noisy cat. Can text, code or a quantized image still be generated the way this chapter generates images, many positions at once, each conditioned on both sides of a gap, with a likelihood bound to train against? Austin et al. (2021) made the noise a Markov chain on the vocabulary and found that the best chain erases tokens, replacing them by a mask symbol. Sahoo et al. (2024) and Shi et al. (2024) showed that the bound of this chain is a weighted form of the masked-language-model loss, the objective of today's diffusion language models (Nie, Zhu, You, et al. 2025).

The variational bound of Section 17.5 carries over unchanged; with erasure as the noise it becomes a cross-entropy on the erased positions. The sampler reveals many tokens per network evaluation, at a cost that two tokens make exact, and we test both on a quantized version of the running example and on binary images.

```
torch.set_float32_matmul_precision('high')  # tf32 matmuls on tensor cores
```

```
from scipy.special import ndtr, xlogy
```

17.8.1 Corrupting Tokens

Tokens need a forward process with closed-form marginals and posteriors that ends in a state carrying no information.

17.8.1.1 Transition Matrices

Let a token take one of K values, written as a one-hot row vector $\mathbf{x} \in \{0, 1\}^K$. Noise for a token is a random change of its value, given by *transition matrices* $\mathbf{Q}_t \in [0, 1]^{K \times K}$ whose rows sum to one: $[\mathbf{Q}_t]_{ij}$ is the probability of moving from value i to value j at step t. Here $t = 1, \dots, T$ counts steps, as k did for the Gaussian chain, and the network receives t/T, the time on the chapter's clock. These are the row-stochastic matrices of Markov chain Monte Carlo (Section 36.8.3), with the same convention that a distribution is a row vector multiplied by $\mathbf{Q}_t$ from the right. Hence

$$q(\mathbf{x}_t \mid \mathbf{x}_{t-1}) = \mathrm{Cat}(\mathbf{x}_t;\, \mathbf{x}_{t-1}\mathbf{Q}_t), \tag{17.58}$$

where $\mathrm{Cat}(\mathbf{x}; \mathbf{p})$ is the categorical distribution with probability vector $\mathbf{p}$, here the row of $\mathbf{Q}_t$ selected by the current value. A sequence of L tokens is corrupted by running the chain at every position independently. Composing two transitions multiplies their matrices, so by induction on t the marginal after t steps is

$$q(\mathbf{x}_t \mid \mathbf{x}_0) = \mathrm{Cat}(\mathbf{x}_t;\, \mathbf{x}_0 \bar{\mathbf{Q}}_t), \qquad \bar{\mathbf{Q}}_t = \mathbf{Q}_1 \mathbf{Q}_2 \cdots \mathbf{Q}_t. \tag{17.59}$$

The posterior of one step given both of its ends follows from Bayes' rule and the Markov property,

$$q(\mathbf{x}_{t-1} \mid \mathbf{x}_t, \mathbf{x}_0) = \frac{q(\mathbf{x}_t \mid \mathbf{x}_{t-1})\, q(\mathbf{x}_{t-1} \mid \mathbf{x}_0)}{q(\mathbf{x}_t \mid \mathbf{x}_0)} = \mathrm{Cat}\left(\mathbf{x}_{t-1};\, \frac{\mathbf{x}_t \mathbf{Q}_t^\top \odot \mathbf{x}_0 \bar{\mathbf{Q}}_{t-1}}{\mathbf{x}_0 \bar{\mathbf{Q}}_t \mathbf{x}_t^\top}\right), \tag{17.60}$$

where $\odot$ is the elementwise product: each possible predecessor is weighted by its probability given the start and by its probability of leading to the observed $\mathbf{x}_t$.

A binary token that flips with probability 0.1 at every step shows the posterior at work. Here $\mathbf{Q}_1 = \mathbf{Q}_2 = \left(\begin{smallmatrix} 0.9 & 0.1 \\ 0.1 & 0.9 \end{smallmatrix}\right)$. Start from the first value, $\mathbf{x}_0 = (1, 0)$, and suppose that after two steps the token shows the second value, $\mathbf{x}_2 = (0, 1)$. This happens with probability $0.9 \cdot 0.1 + 0.1 \cdot 0.9 = 0.18$, the denominator. For the middle state, $\mathbf{x}_2 \mathbf{Q}_2^\top = (0.1, 0.9)$ and $\mathbf{x}_0 \bar{\mathbf{Q}}_1 = (0.9, 0.1)$. Their elementwise product is $(0.09, 0.09)$, and dividing by 0.18 gives $(1/2, 1/2)$. The token flipped once, and given both ends the flip is equally likely to have happened at the first step or at the second.

17.8.1.2 The Absorbing Kernel

The bound needs a closed form for $\bar{\mathbf{Q}}_t$, and the sampler a known starting state. The *absorbing kernel* of Austin et al. (2021) provides both. It enlarges the vocabulary by one value, the *mask* $[\mathrm{M}]$, and moves each token there with probability β_t at step t; a masked token stays masked:

$$\mathbf{Q}_t = (1 - \beta_t)\, \mathbf{I} + \beta_t\, \mathbf{1} \mathbf{e}_{\mathrm{M}}^\top, \tag{17.61}$$

with $\mathbf{1}$ the all-ones column vector and $\mathbf{e}_{\mathrm{M}}$ the one-hot column vector of the mask, so that every row of $\mathbf{1}\mathbf{e}_{\mathrm{M}}^\top$ puts all its mass on the mask.

Products of such matrices collapse. Write $\mathbf{P} = \mathbf{1}\mathbf{e}_{\mathrm{M}}^\top$. Since $\mathbf{e}_{\mathrm{M}}^\top \mathbf{1} = 1$, we have $\mathbf{P}^2 = \mathbf{P}$, and therefore $(a\mathbf{I} + b\mathbf{P})(c\mathbf{I} + d\mathbf{P}) = ac\,\mathbf{I} + (ad + bc + bd)\,\mathbf{P}$. When $a + b = c + d = 1$ the second coefficient equals $1 - ac$, so the coefficient of $\mathbf{I}$ multiplies along the chain. Let $\bar{\alpha}_t = \prod_{s \le t} (1 - \beta_s)$, the same product as for the Gaussian chain of Section 17.2. Here it has a direct meaning, the probability that a token survives t steps without being masked. Then

$$\bar{\mathbf{Q}}_t = \bar{\alpha}_t\, \mathbf{I} + (1 - \bar{\alpha}_t)\, \mathbf{1} \mathbf{e}_{\mathrm{M}}^\top. \tag{17.62}$$

The chain never confuses one token with another; it only erases. A schedule with $\bar{\alpha}_T = 0$ ends at the fully masked sequence, which carries no information about the data, as the Gaussian chain ends at pure noise (Figure 17.15). The other common choice, the *uniform kernel*, resamples a token uniformly from the vocabulary (Hoogeboom et al. 2021) (Exercise 1).

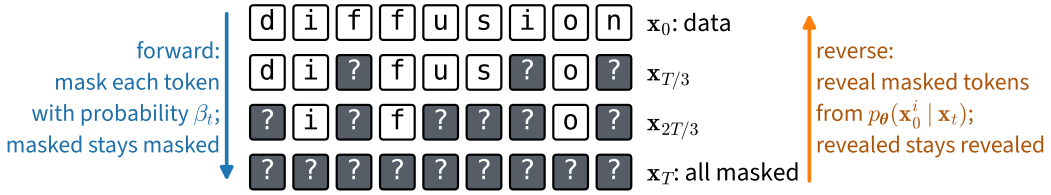


Figure 17.15: The absorbing chain on nine tokens. Forward, each token is independently replaced by the mask with probability β_t per step and stays masked; after T steps every token is masked. Reverse, the model predicts masked tokens from the partially masked sequence, and a revealed token is never changed.

We use the absorbing kernel from now on: it gave the best likelihoods on text in the comparisons of Austin et al., and its posterior is the simplest.

17.8.2 The Masked Bound

17.8.2.1 The Same Bound, a Different Target

The reverse model is a Markov chain $p_\theta(\mathbf{x}_{t-1} \mid \mathbf{x}_t)$, started from the all-mask sequence. Section 17.5 bounded the negative log-likelihood of such a chain by

$$-\log p_\theta(\mathbf{x}_0) \le L_{\text{VLB}} = D_{\text{KL}}(q(\mathbf{x}_T \mid \mathbf{x}_0) \,\|\, p(\mathbf{x}_T)) + \sum_{t=2}^{T} \mathbb{E}_q \Big[D_{\text{KL}}(q(\mathbf{x}_{t-1} \mid \mathbf{x}_t, \mathbf{x}_0) \,\|\, p_\theta(\mathbf{x}_{t-1} \mid \mathbf{x}_t)) \Big] + \mathbb{E}_q [-$$

the evidence lower bound of Section 36.7 with the forward chain as the variational distribution. Its derivation used only the Markov property and Bayes' rule, so it holds for tokens as it stands.

What changes is the target. For Gaussian noise the network predicts the noise or the clean point; for tokens the natural target is the clean token itself. Following Austin et al. (2021), the network outputs, for each position i, a distribution $p_\theta(\mathbf{x}_0^i \mid \mathbf{x}_t)$ over the clean values given the whole corrupted sequence, and the reverse transition averages the true posterior over this prediction,

$$p_\theta(\mathbf{x}_{t-1} \mid \mathbf{x}_t) = \sum_{\hat{\mathbf{x}}_0} q(\mathbf{x}_{t-1} \mid \mathbf{x}_t, \hat{\mathbf{x}}_0)\, p_\theta(\hat{\mathbf{x}}_0 \mid \mathbf{x}_t), \tag{17.63}$$

position by position, with the prediction at an unmasked position taken to be the observed token. A perfect prediction makes this the true reverse transition. Austin et al. renormalize the average differently; for the absorbing kernel the two forms agree (Exercise 5).

17.8.2.2 The Posterior of the Absorbing Chain

Proposition (posterior of the absorbing chain). *Under the absorbing kernel with $\beta_1 > 0$, so that $\bar{\alpha}_t < 1$ for every $t \ge 1$, consider one position with clean value $\mathbf{x}_0 \ne [M]$. If $\mathbf{x}_t = \mathbf{x}_0$, then $\mathbf{x}_{t-1} = \mathbf{x}_0$ with probability one. If $\mathbf{x}_t = [M]$, then*

$$q(\mathbf{x}_{t-1} = \mathbf{x}_0 \mid \mathbf{x}_t = [\mathrm{M}], \mathbf{x}_0) = \frac{\bar{\alpha}_{t-1} - \bar{\alpha}_t}{1 - \bar{\alpha}_t}, \qquad q(\mathbf{x}_{t-1} = [\mathrm{M}] \mid \mathbf{x}_t = [\mathrm{M}], \mathbf{x}_0) = \frac{1 - \bar{\alpha}_{t-1}}{1 - \bar{\alpha}_t}, \tag{17.64}$$

and no other value has positive probability.

Proof. Evaluate Equation 17.60 with Equation 17.61 and Equation 17.62. The vector $\mathbf{x}_0 \bar{\mathbf{Q}}_{t-1}$ puts mass $\bar{\alpha}_{t-1}$ on $\mathbf{x}_0$ and $1-\bar{\alpha}_{t-1}$ on the mask, so only these two predecessors are possible. If $\mathbf{x}_t = \mathbf{x}_0$, the vector $\mathbf{x}_t \mathbf{Q}_t^\top$ has the entry $1 - \beta_t$ at $\mathbf{x}_0$ and the entry 0 at the mask, because a mask never returns to a token, and the posterior is a point mass. If $\mathbf{x}_t = [\mathrm{M}]$, the entries are β_t at $\mathbf{x}_0$ and 1 at the mask, so the unnormalized posterior is $\bar{\alpha}_{t-1}\beta_t$ on $\mathbf{x}_0$ and $1 - \bar{\alpha}_{t-1}$ on the mask. Their sum, $1 - \bar{\alpha}_{t-1}(1 - \beta_t) = 1 - \bar{\alpha}_t$, is the denominator $q(\mathbf{x}_t = [\mathrm{M}] \mid \mathbf{x}_0)$, and $\bar{\alpha}_{t-1}\beta_t = \bar{\alpha}_{t-1} - \bar{\alpha}_t$. ■

One step back, a masked token either stays masked or reverts to its clean value. The reverse model Equation 17.63 therefore copies the unmasked positions of $\mathbf{x}_t$ and replaces each masked one, independently, by a draw from $p_\theta(\mathbf{x}_0^i \mid \mathbf{x}_t)$ with the probability of reverting.

17.8.2.3 A Weighted Cross-Entropy

Proposition (masked diffusion bound). *For the absorbing kernel with $\bar{\alpha}_0 = 1$, $\bar{\alpha}_t < 1$ for $t \geq 1$, and $\bar{\alpha}_T = 0$, and the reverse model just described,*

$$L_{\mathrm{VLB}} = \sum_{t=1}^{T} \frac{\bar{\alpha}_{t-1} - \bar{\alpha}_t}{1 - \bar{\alpha}_t} \, \mathbb{E}_{q(\mathbf{x}_t \mid \mathbf{x}_0)} \Big[\sum_{i \,:\, \mathbf{x}_t^i = [\mathrm{M}]} - \log p_\theta(\mathbf{x}_0^i \mid \mathbf{x}_t) \Big]. \tag{17.65}$$

Proof. The prior term vanishes: $\bar{\alpha}_T = 0$ makes $q(\mathbf{x}_T \mid \mathbf{x}_0)$ the point mass on the all-mask sequence, which is also $p(\mathbf{x}_T)$. Both $q(\mathbf{x}_{t-1} \mid \mathbf{x}_t, \mathbf{x}_0)$ and $p_\theta(\mathbf{x}_{t-1} \mid \mathbf{x}_t)$ factorize over positions, so each divergence is a sum of per-position divergences. At an unmasked position both distributions are the same point mass and the divergence is zero. At a masked position write $w_t = (\bar{\alpha}_{t-1} - \bar{\alpha}_t)/(1 - \bar{\alpha}_t)$. The true posterior puts $1 - w_t$ on the mask and w_t on $\mathbf{x}_0^i$; the model puts $1 - w_t$ on the mask and $w_t \, p_\theta(\cdot \mid \mathbf{x}_t)$ on the tokens; hence

$$D_{\mathrm{KL}} = (1 - w_t) \log \frac{1 - w_t}{1 - w_t} + w_t \log \frac{w_t}{w_t \, p_\theta(\mathbf{x}_0^i \mid \mathbf{x}_t)} = -w_t \log p_\theta(\mathbf{x}_0^i \mid \mathbf{x}_t).$$

This accounts for $t = 2, \ldots, T$. The reconstruction term is $\mathbb{E}[-\log p_\theta(\mathbf{x}_0 \mid \mathbf{x}_1)]$. Since $w_1 = (1 - \bar{\alpha}_1)/(1 - \bar{\alpha}_1) = 1$, the model replaces every position masked at step 1 by a draw from $p_\theta(\cdot \mid \mathbf{x}_1)$ and copies the rest, so this term equals the sum of $-\log p_\theta(\mathbf{x}_0^i \mid \mathbf{x}_1)$ over the masked positions, which is the $t = 1$ term of the sum. ■

Divided by $\log 2$ and the number of tokens, the bound is a code length in bits per token (Section 37.2), and its exponential in nats per token bounds the perplexity (Section 37.2.4.1).

Under the *linear schedule* $\bar{\alpha}_t = 1 - t/T$, a fraction t/T of the tokens is masked at step t, and $w_t = (1/T)/(t/T) = 1/t$. An error on a heavily masked sequence is charged less than the same error on a lightly masked one. Since the number of masked positions grows in proportion to t, every step carries the same expected weighted count of masked positions, L/T for L tokens; the steps differ only in how hard their masked tokens are to predict.

The sum is also a Riemann sum. Write $\ell(\mathbf{x}_0, \mathbf{x})$ for the cross-entropy on the masked positions of $\mathbf{x}$, the inner sum of Equation 17.65, and let $T \to \infty$ with $\bar{\alpha}_t = \bar{\alpha}(t/T)$ for a decreasing function on the chapter's clock, data at time 0 and the all-mask sequence at time 1. Then $w_t \approx -\bar{\alpha}'(\tau)/(T(1-\bar{\alpha}(\tau)))$ at $\tau = t/T$, and the substitution $u = 1 - \bar{\alpha}(\tau)$, the masking probability, turns the limit of the bound into

$$\int_0^1 \frac{-\bar{\alpha}'(\tau)}{1-\bar{\alpha}(\tau)}\, \mathbb{E}\big[\ell(\mathbf{x}_0, \mathbf{x}_\tau)\big]\, d\tau = \int_0^1 \frac{1}{u}\, \mathbb{E}_u\big[\ell(\mathbf{x}_0, \mathbf{x})\big]\, du,$$

where $\mathbb{E}_u$ masks each token independently with probability u. For a network told the masking probability rather than the time, the limit does not depend on the schedule, the discrete counterpart of the schedule invariance of the Gaussian bound (Section 17.5) (Kingma et al. 2021).

The function `masked_loss` estimates the bound with one step t per sequence, given the schedule `alpha_bar` (the values $\bar{\alpha}_0, \dots, \bar{\alpha}_T$) and the index `mask_id` of the mask; the factor T makes a uniformly drawn step unbiased for the sum over steps. The network receives t/T, the time on the chapter's clock, and the sum runs over every axis after the first, so pairs of tokens and images share the function.

```
def masked_loss(model, x0, t, alpha_bar, mask_id):
    """One-draw estimate of the masked bound per sequence, in nats."""
    T = len(alpha_bar) - 1
    shape = (-1,) + (1,) * (x0.dim() - 1)      # broadcast over positions
    masked = (torch.rand(x0.shape, device=x0.device)
              < (1 - alpha_bar[t]).view(shape))  # with prob. 1 - abar_t
    logits = model(torch.where(masked, mask_id, x0), t / T)
    nll = -torch.log_softmax(logits, -1).gather(-1, x0[..., None])[..., 0]
    w = (alpha_bar[t - 1] - alpha_bar[t]) / (1 - alpha_bar[t])  # 1/t if linear
    return (T * w.view(shape) * nll * masked).flatten(1).sum(1)
```

17.8.3 Sampling by Unmasking

17.8.3.1 Parallel Reveals and the Independence Error

At step t, one forward pass yields $p_\theta(\cdot \mid \mathbf{x}_t)$ at every masked position. Each masked position draws a candidate token and accepts it with probability w_t. As for the Gaussian samplers of Section 17.4, fewer steps are possible: choose levels $T = t_0 > t_1 > \cdots > t_S = 0$. The coarser chain has the same marginals Equation 17.62, so the proposition applies to it unchanged, and a jump from t_k to t_{k+1} reveals each masked token with probability $(\bar{\alpha}_{t_{k+1}} - \bar{\alpha}_{t_k})/(1-\bar{\alpha}_{t_k})$. The function `unmask` returns the list of states, the last of which is the sample.

```
@torch.inference_mode()
def unmask(model, shape, steps, alpha_bar, mask_id):
    """Reveal tokens from the all-mask state in `steps` jumps; all states."""
    T = len(alpha_bar) - 1
    levels = torch.linspace(T, 0, steps + 1).round().long().tolist()
    states = [torch.full(shape, mask_id, device=alpha_bar.device)]
    for t, s in zip(levels[:-1], levels[1:]):
```

```
            z = states[-1]
            logits = model(z, torch.full(shape[:1], t / T, device=z.device))
            draw = torch.distributions.Categorical(logits=logits).sample()
            p = (alpha_bar[s] - alpha_bar[t]) / (1 - alpha_bar[t])  # reveal prob.
            reveal = (z == mask_id) & (torch.rand(shape, device=z.device) < p)
            states.append(torch.where(reveal, draw, z))
        return states
```

Tokens revealed in the same jump are drawn independently, each from its own predicted distribution, because the network's prediction is a product over positions; tokens revealed in different jumps are drawn one after another, each conditioned on those already revealed. Two tokens show the cost exactly. With S equally spaced levels under the linear schedule, a token is revealed in any given jump with probability $1/S$, so both tokens are revealed in the same jump with probability $S \cdot (1/S)^2 = 1/S$. Take an *exact* network, whose prediction at each masked position is the true conditional of that token given the unmasked ones. Its sampler draws the pair from the mixture

$$\left(1 - \tfrac{1}{S}\right) p + \tfrac{1}{S}\, p^1 \otimes p^2$$

of the joint distribution p of the two tokens and the product of its marginals p^1 and p^2. In total variation, half the summed absolute difference of two probability tables (Section 37.3.3), the mixture lies $\mathrm{TV}(p^1 \otimes p^2, p)/S$ from p: the error falls as one over the number of steps and grows with the dependence between the tokens.

The chapter's running example makes this concrete: the mixture of three Gaussians in the plane with weights 0.5, 0.3 and 0.2 and standard deviation 0.5 (Section 17.1). Quantizing the plane onto a 32×32 grid over $[-5, 5]^2$ turns each sample into a pair of tokens, the indices of its cell along the two axes. Cell probabilities factorize into one-dimensional Gaussian integrals, so the cell below computes the true table p exactly, with its entropy, the mutual information of the two tokens, and the error of an exact network's S-step sampler.

```
mix = d2l.GaussianMixture.three_modes()
means, weights = np.float64(mix.means), np.float64(mix.weights)
K, T = 32, 32                 # vocabulary of each token; number of steps
edges = np.linspace(-5, 5, K + 1)
cdf = ndtr((edges - means[:, :, None]) / mix.std)  # (modes, axes, K + 1)
cdf[:, :, 0], cdf[:, :, -1] = 0, 1      # edge cells extend to infinity
cell = np.diff(cdf, axis=2)             # mass of each cell, per mode, axis
table = np.einsum('m,mi,mj->ij', weights, cell[:, 0], cell[:, 1])
p1, p2 = table.sum(1), table.sum(0)     # marginals of the two tokens
product = np.outer(p1, p2)
H = lambda p: -xlogy(p, p).sum()        # entropy in nats
tv = lambda p, q: np.abs(p - q).sum() / 2
mutual = H(p1) + H(p2) - H(table)
print(f'entropy {H(table):.3f} nats, mutual information {mutual:.3f} nats')
for S in (1, 2, 8, 32):
    print(f'{S:>2} steps, exact network: TV {tv(product, table) / S:.3f}')
```

```
entropy 4.840 nats, mutual information 0.479 nats
 1 steps, exact network: TV 0.314
 2 steps, exact network: TV 0.157
```

```
 8 steps, exact network: TV 0.039
32 steps, exact network: TV 0.010
```

17.8.3.2 Diffusion Language Models in Practice

Practical diffusion language models change two parts of this recipe. The first is the network's input. Which positions are masked is decided by coin flips that never look at the tokens, so the step only says how many masks to expect, which the input already shows. The best prediction at a masked position therefore does not depend on t (Ou et al. 2025), and the denoiser is a bidirectional transformer, a GPT without the causal mask, with one extra vocabulary entry for $[\mathrm{M}]$ (Nie, Zhu, You, et al. 2025).

The second is the order of reveals. MaskGIT (Chang et al. 2022) and LLaDA draw a candidate at every masked position, keep the candidates that are most probable under their own predicted distributions, and mask the others again. The two-token table shows the effect without training. For two tokens, *confidence remasking* draws both from their marginals, keeps the draw with the higher probability, and redraws the other from its conditional given the kept one. The cell compares its exact output table with those of one step, two random jumps, and a fixed order (first token from its marginal, second from its conditional).

```
c2 = table / p1[:, None]                # p(x2 | x1); each row sums to one
c1 = table / p2[None, :]                # p(x1 | x2); each column sums to one
keep1 = p1[:, None] >= p2[None, :]      # the draw of x1 is the likelier one
confidence = ((product * keep1).sum(1)[:, None] * c2
              + (product * ~keep1).sum(0)[None, :] * c1)
centers = (edges[:-1] + edges[1:]) / 2
grid = np.stack(np.meshgrid(centers, centers, indexing='ij'), -1)
mode = ((grid[:, :, None] - means) ** 2).sum(-1).argmin(-1)  # nearest mode
orders = {'one step': product, 'two random jumps': (product + table) / 2,
          'fixed order': p1[:, None] * c2, 'confidence': confidence}
print(f'{"order":>16} {"TV":>6} {"entropy":>8}  mass near the modes')
for name, q in orders.items():
    masses = ' '.join(f'{q[mode == k].sum():.3f}' for k in range(3))
    print(f'{name:>16} {tv(q, table):6.3f} {H(q):8.3f}  {masses}')
```

```
           order     TV  entropy  mass near the modes
        one step  0.314    5.319  0.482 0.321 0.197
two random jumps  0.157    5.216  0.491 0.310 0.199
     fixed order  0.000    4.840  0.500 0.300 0.200
      confidence  0.179    4.406  0.606 0.322 0.072
```

With exact conditionals, any order that reveals one token per step is exact, as the fixed order shows. Two random jumps reveal both tokens together half of the time and halve the one-step error. Confidence remasking also reveals one token per step, yet its error is larger than that of random jumps. Which draw it keeps depends on the drawn values, and it favors values that were likely to begin with, so it seeks modes: the smallest mode falls from 0.2 to about 0.07, and the entropy drops from 4.84 to 4.41 nats. On text, lower entropy often reads as fluency, and with an imperfect network it can help to reveal the easy positions first. But the bound does not describe this sampler and cannot detect its bias. In all of these samplers a revealed token is never revised, so an early mistake stays.

At equal compute on fresh data, autoregressive models reach a lower loss (Prabhudesai et al. 2025), so the case for diffusion is not a better likelihood but parallel decoding, many tokens

per network evaluation, and infilling, generation conditioned on text on both sides of a gap. For images, masked diffusion works on the discrete codes of a learned tokenizer (Chang et al. 2022); the binary pixels below stand in for such codes because they need no tokenizer.

17.8.4 Experiments

17.8.4.1 The Quantized Mixture

The network for the pair takes the one-hot encoding of both tokens, with the mask as a thirty-third value at index K, together with the time t/T, and outputs logits over the 32 clean values at each position. The schedule is linear with $T = 32$ steps.

```
class MaskedMLP(nn.Module):
    """Logits of both clean tokens of a pair, given the masked pair and t."""
    def __init__(self, K, hidden=256):
        super().__init__()
        self.K = K
        self.net = d2l.make_mlp([2 * (K + 1) + 1, hidden, hidden, 2 * K],
                                nn.SiLU())

    def forward(self, z, t):  # z: (n, 2) in {0, ..., K}, where K is the mask
        h = F.one_hot(z, self.K + 1).float().flatten(1)
        return self.net(torch.cat([h, t[:, None]], 1)).view(-1, 2, self.K)

def quantize(x):  # points in the plane to pairs of tokens in {0, ..., K - 1}
    return ((x + 5) / 10 * K).floor().long().clamp(0, K - 1)

alpha_bar = 1 - torch.arange(T + 1) / T     # the linear schedule
```

We train it with `d2l.train_regression` (Section 17.2) on fresh draws from the mixture, 512 pairs per batch, each with a uniformly drawn step, with a learning rate that decays along a cosine over 6,000 updates.

```
torch.manual_seed(0)
batch_fn = lambda: (quantize(mix.sample(512)), torch.randint(1, T + 1, (512,)))
loss_fn = lambda model, batch: masked_loss(model, *batch, alpha_bar,
                                           mask_id=K).mean()
_, pair_model, _ = d2l.train_regression(MaskedMLP(K), loss_fn, batch_fn,
                                        6000, cosine=True)
```

With two positions, the expectation in the bound can be computed exactly. At each step there are three mask patterns with at least one mask, a pattern with m masks has probability $u^m(1-u)^{2-m}$ with $u = 1-\bar{\alpha}_t$, and the 1,024 pairs are weighted by the table. This average cannot fall below the entropy of the table, the smallest average negative log-likelihood any model attains (Section 37.1). Even an exact network stays above it, because it predicts two masked tokens independently and the bound charges the mutual information (Section 37.6) for this. An exact network attains

$$H(X^1, X^2) + \frac{I(X^1; X^2)}{T},$$

where X^1 and X^2 are the two tokens of a pair drawn from the table (Exercise 7). This floor, $4.840 + 0.479/32 \approx 4.855$ nats, binds every network that predicts the positions independently.

```
@torch.inference_mode()
def exact_bound(model, table, alpha_bar, mask_id):
    """The bound averaged over the table, summed exactly over t and masks."""
    T, K = len(alpha_bar) - 1, len(table)
    pairs = torch.cartesian_prod(torch.arange(K), torch.arange(K))
    p, total = torch.tensor(table.flatten(), dtype=torch.float32), 0.0
    for t in range(1, T + 1):
        u = 1 - alpha_bar[t]
        w = (alpha_bar[t - 1] - alpha_bar[t]) / u
        for masked in torch.tensor([[1, 0], [0, 1], [1, 1]], dtype=torch.bool):
            z, m = torch.where(masked, mask_id, pairs), masked.sum()
            logp = torch.log_softmax(model(z, torch.full((K * K,), t / T)), -1)
            nll = -(logp.gather(-1, pairs[..., None])[..., 0] * masked).sum(1)
            total += u ** m * (1 - u) ** (2 - m) * w * (p @ nll)
    return float(total)

bound = exact_bound(pair_model, table, alpha_bar, mask_id=K)
print(f'exact bound {bound:.3f} nats; '
      f'entropy {H(table):.3f}, floor {H(table) + mutual / T:.3f}')
```

```
exact bound 4.856 nats; entropy 4.840, floor 4.855
```

The bound comes within a few hundredths of a nat of the entropy: the model has learned the marginal of each token and its conditional given the other. Almost all of the gap is the floor; the network's own excess over the floor is of the order of a thousandth of a nat. The floor is not just looseness of the bound: the thirty-two-step sampler of an exact network models $(1 - 1/32)\, p + (1/32)\, p^1 \otimes p^2$ rather than p. The next cell compares histograms of 20,000 pairs drawn with $S \in \{32, 8, 2, 1\}$ steps with the table, after printing the floor that 20,000 true samples impose.

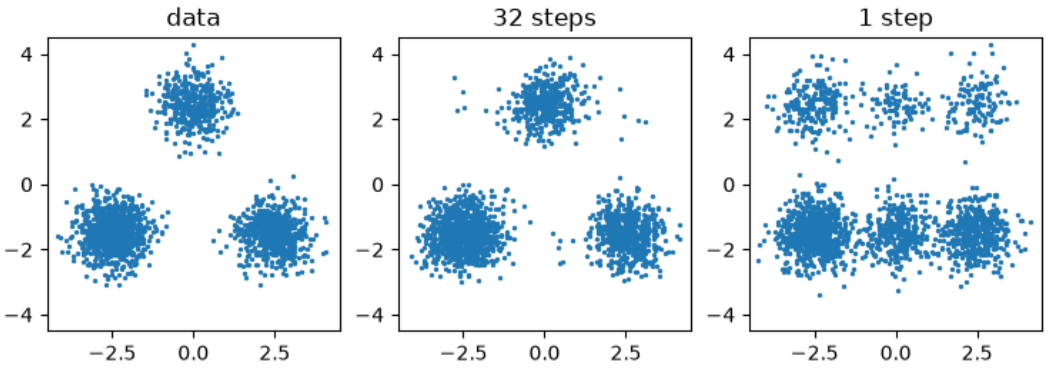
```
def histogram(z):  # pairs of tokens to a K x K table of frequencies
    counts = torch.bincount(z[:, 0] * K + z[:, 1], minlength=K * K)
    return np.float64(counts.view(K, K)) / len(z)

torch.manual_seed(1)
data = quantize(mix.sample(20000))
print(f'TV(histogram of 20000 true samples, table): '
      f'{tv(histogram(data), table):.3f}')
samples = {S: unmask(pair_model, (20000, 2), S, alpha_bar, mask_id=K)[-1]
           for S in (32, 8, 2, 1)}
for S, z in samples.items():
    print(f'{S:>2} steps: TV {tv(histogram(z), table):.3f}, '
          f'exact network {tv(product, table) / S:.3f}')
```

```
TV(histogram of 20000 true samples, table): 0.037
32 steps: TV 0.050, exact network 0.010
 8 steps: TV 0.067, exact network 0.039
 2 steps: TV 0.171, exact network 0.157
 1 steps: TV 0.326, exact network 0.314
```

The distance tracks $\mathrm{TV}(p^1 \otimes p^2, p)/S$. At thirty-two and eight steps the sampling floor and the model error add a few hundredths; at two steps and one step the systematic error dominates, and the distance lies within a few hundredths of the prediction. In the panels (2,000 pairs, placed at random inside their cells), the one-step sampler puts mass where the column of one mode crosses the row of another, where the data has none.

```
decode = lambda z: (z[:2000] + torch.rand(2000, 2)) / K * 10 - 5  # in cell
d2l.scatter_panels([decode(data), decode(samples[32]), decode(samples[1])],
                   ['data', '32 steps', '1 step']);
```



17.8.4.2 Images as Binary Tokens

Thresholding Fashion-MNIST at half intensity makes every pixel a token with two values, and a masked pixel has a third. An image is then a sequence of 784 tokens on a grid, so the network can be the chapter's U-Net (Section 17.3); it takes the one-hot encoding as three input channels and returns the logits of the clean pixel as two output channels. The loss and the sampler are those of the pair, with a linear schedule of $T = 100$ steps and the mask at index 2.

```
X, _, X_test, _ = d2l.load_fashion_mnist_pm1(binary=True)  # (n, 28, 28)
T_img = 100                                    # steps of the image chain
alpha_bar_img = 1 - torch.arange(T_img + 1, device=X.device) / T_img

class MaskedUNet(nn.Module):
    """d2l.UNet on one-hot pixels (black, white, mask); clean-pixel logits."""
    def __init__(self):
        super().__init__()
        self.unet = d2l.UNet(in_channels=3, out_channels=2)

    def forward(self, z, t):  # z: (n, 28, 28) in {0, 1, 2}
        x = F.one_hot(z, 3).float().permute(0, 3, 1, 2)
        return self.unet(x, t).permute(0, 2, 3, 1)  # (n, 28, 28, 2)
```

Training takes 6,000 updates on batches of 128 images, with a constant learning rate and a moving average of the weights with a half-life of 1,000 updates.

```
torch.manual_seed(2)
def batch_fn():
    i = torch.randint(0, len(X), (128,), device=X.device)
    return X[i], torch.randint(1, T_img + 1, (128,), device=X.device)

loss_fn = lambda model, batch: masked_loss(  # per pixel
    model, *batch, alpha_bar_img, mask_id=2).mean() / (28 * 28)
_, unet, _ = d2l.train_regression(MaskedUNet().to(X.device), loss_fn,
                                  batch_fn, 6000)
```

We estimate the bound on 1,000 held-out images, with one random mask per image and level, and compare it with the cross-entropy of a model that treats pixels as independent, using the frequency of white at each position in the training set.

```
X_eval = X_test[:1000]
with torch.inference_mode():
    bound = sum(masked_loss(unet, X_eval, torch.full_like(X_eval[:, 0, 0], t),
                            alpha_bar_img, mask_id=2).mean()
                for t in range(1, T_img + 1)) / T_img
white = X.float().mean(0).clamp(1e-6, 1 - 1e-6)  # frequency of white
indep = -torch.where(X_eval == 1, white.log(), (1 - white).log()).sum((1, 2))
bits = lambda nats: f'{nats / math.log(2) / (28 * 28):.3f} bits per pixel'
print(f'bound {bits(bound)}, independent pixels {bits(indep.mean())}')
```

```
bound 0.171 bits per pixel, independent pixels 0.702 bits per pixel
```

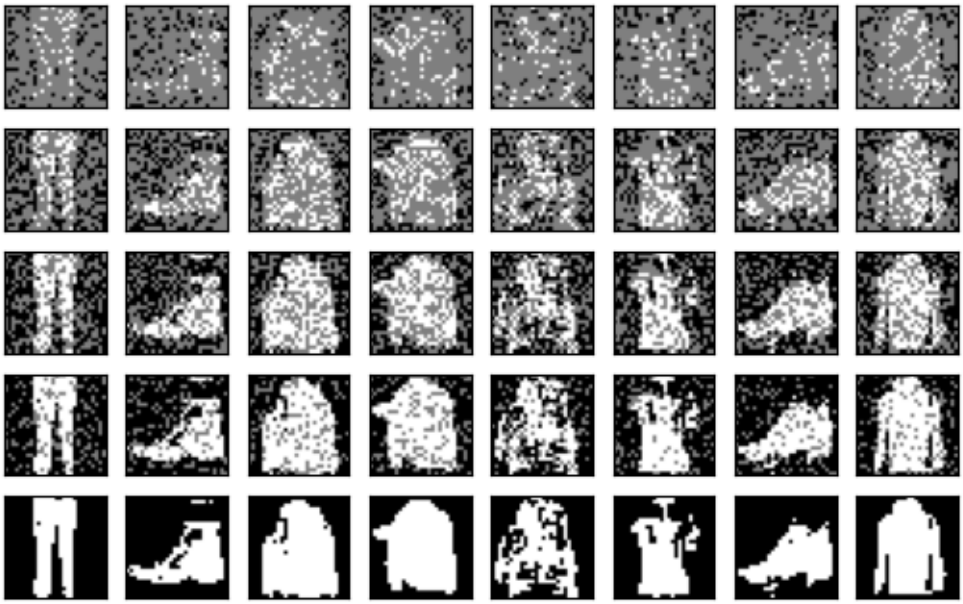
The bound is about 0.17 bits per pixel, against about 0.70 for independent pixels. As code lengths (Section 37.2), the model could encode a held-out garment in about 135 bits, where coding each pixel on its own takes about 550; the model has captured much of the dependence between pixels. The first grid shows 48 samples drawn in a hundred steps.

```
show_tokens = lambda z, rows: d2l.show_gray(  # masked pixels in gray
    torch.where(z == 2, 0.0, 2.0 * z - 1)[:, None], rows, 8, scale=0.8)
torch.manual_seed(3)
states = unmask(unet, (48, 28, 28), T_img, alpha_bar_img, mask_id=2)
show_tokens(states[-1], 6);
```



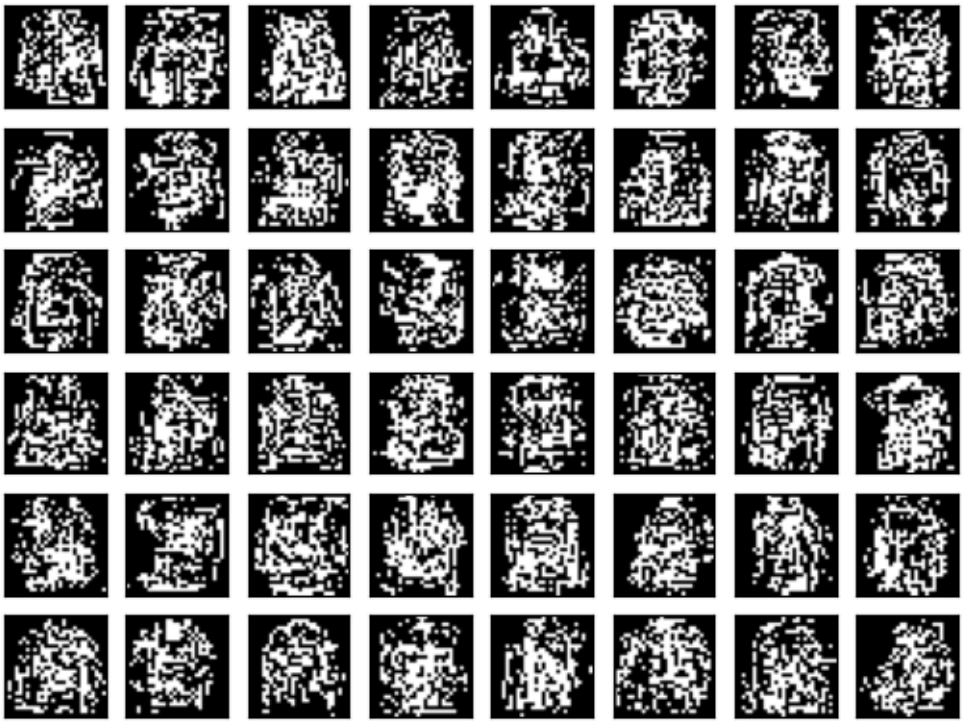
Many samples are recognizable binary garments, rougher than the gray-level samples of the earlier sections; others are fragments, at 6,000 updates of training. The second grid follows eight of them after 20, 40, 60, 80 and 100 steps, with masked pixels in gray.

```
show_tokens(torch.cat([states[k][:8] for k in (20, 40, 60, 80, 100)]), 5);
```



The outline of a garment appears early, and later reveals fill it in consistently, because every prediction is conditioned on everything revealed so far. The third grid uses four steps, each revealing about a quarter of the pixels from predictions that are independent across positions.

```
few = unmask(unet, (48, 28, 28), 4, alpha_bar_img, mask_id=2)[-1]
show_tokens(few, 6);
```



No garment survives: each step reveals about 200 pixels drawn independently of one another, and the result is texture.

To measure what four steps lose, we count *speckles*, pixels that differ from all four of their neighbors.

```
def speckles(z):  # fraction of interior pixels unlike all four neighbors
    c = z[:, 1:-1, 1:-1]
    return ((c != z[:, :-2, 1:-1]) & (c != z[:, 2:, 1:-1])
            & (c != z[:, 1:-1, :-2]) & (c != z[:, 1:-1, 2:])).float().mean()

print(f'speckles: data {speckles(X[:1000]):.4f}, 100 steps '
      f'{speckles(states[-1]):.4f}, 4 steps {speckles(few):.4f}')
```

```
speckles: data 0.0037, 100 steps 0.0059, 4 steps 0.0351
```

The data has about four speckles per thousand interior pixels, the hundred-step samples about twice as many, and the four-step samples about nine times as many. The count is not a measure of sample quality; it measures the independence error, the image counterpart of the crossed modes in the one-step panel of the pair.

17.8.5 Masked Language Models as One Step

Trained at a single masking level, Equation 17.65 is the objective of BERT (Devlin et al. 2018), which hides 15% of the tokens of a text and predicts them from both sides (Section 43.9), up to BERT's replacement of some masks by random or unchanged tokens. A masked language model is thus a discrete diffusion model with one step (Austin et al. 2021), though one that

has seen only one masking level, so a sampler starting from the all-mask sequence would show it inputs far from its training data. Reveal exactly one token per step in a random order, and the sampler is an order-agnostic autoregressive model (Uria et al. 2014). Fix the order to left to right, and it is the language model of Section 11.2, whose conditionals are the predictions at inputs with an unmasked prefix and a masked rest. Diffusion language models operate between these two extremes, trading sequential network evaluations for the independence error; the next part of this book studies masked and autoregressive language models at scale. Over the chapter the four limits of the game of Chapter 16 found their answers: a density and a bound (Section 17.5), one regression in place of two networks (Section 17.2), a likelihood under which a dropped mode costs on every one of its points, and a transformer that improves steadily as it grows (Section 17.6).

17.8.6 Summary

Diffusion does not need Gaussian noise. A Markov chain on a vocabulary has closed-form marginals and posteriors, and the variational bound applies to it as it stands; with the absorbing kernel it becomes a cross-entropy on the masked positions, weighted by $1/t$ under the linear schedule. Sampling reveals masked tokens and never revises them; tokens revealed in the same step are drawn independently, at an error that falls as one over the number of steps. On the quantized mixture the bound came within a few hundredths of a nat of the entropy, and a U-Net on binary Fashion-MNIST reached about 0.17 bits per pixel. BERT is the one-step case of this model, and a left-to-right language model the case that reveals one token per step.

17.8.7 Exercises

1. **The uniform kernel.** Let the forward chain use the uniform kernel $\mathbf{Q}_t = (1-\beta_t)\,\mathbf{I} + \beta_t\,\frac{1}{K}\mathbf{1}\mathbf{1}^\top$ with $\beta_t > 0$ at every step.
    1. Show that $\bar{\mathbf{Q}}_t = \bar{\alpha}_t\,\mathbf{I} + (1-\bar{\alpha}_t)\,\frac{1}{K}\mathbf{1}\mathbf{1}^\top$ and that $\mathbf{x}_t = \mathbf{x}_0$ with probability $\bar{\alpha}_t + (1-\bar{\alpha}_t)/K$. A token that shows a value at step t may therefore have been resampled. Which question about every position does a network have to answer under this kernel but not under the absorbing kernel?
    2. Derive the posterior $q(\mathbf{x}_{t-1} \mid \mathbf{x}_t, \mathbf{x}_0)$ from Equation 17.60, treating the cases $\mathbf{x}_t = \mathbf{x}_0$ and $\mathbf{x}_t \neq \mathbf{x}_0$ separately. Show that for $t \geq 2$ every one of the K values has positive probability, unlike under the two-point posterior Equation 17.64.
    3. The divergence terms of the bound compare this posterior with the reverse model at every position. Why are they more expensive to evaluate than the summand of Equation 17.65?
2. **Weights of other schedules.** The weight of step t in Equation 17.65 is $w_t = (\bar{\alpha}_{t-1} - \bar{\alpha}_t)/(1-\bar{\alpha}_t)$.
    1. Compute w_t for the schedule $\bar{\alpha}_t = 1 - (t/T)^2$ and compare it with the weight $1/t$ of the linear schedule.
    2. Show that a T-step sampling run from the all-mask sequence reveals a given token at step t with probability $\bar{\alpha}_{t-1} - \bar{\alpha}_t$, independently of the other positions, and that this probability equals w_t times the probability that the token is masked at step t of the bound. In which part of a sampling run does the schedule of (a) reveal most of the tokens?

3. For a network that receives the masking probability u, the continuous-time limit in Section 17.8.2.3 is $\int_0^1 \frac{1}{u}\,\mathbb{E}_u[\ell]\,du$ for every schedule. Show that the integrand stays bounded as $u \to 0$, although the weight $1/u$ diverges.
4. Training draws the time τ uniformly from $[0, 1]$. Which density does the masking probability $u = 1-\bar{\alpha}(\tau)$ then have? Why does the schedule change the variance of the one-sample estimate of the bound, although it leaves the limiting bound unchanged (Kingma et al. 2021)?

3. **Pairs revealed together.** Let the sampler use the levels $T = t_0 > t_1 > \cdots > t_S = 0$ under a schedule with $\bar{\alpha}_0 = 1$ and $\bar{\alpha}_T = 0$, and write $\Delta_k = \bar{\alpha}_{t_{k+1}} - \bar{\alpha}_{t_k}$ for the increase of $\bar{\alpha}$ in jump k, where $k = 0, \dots, S-1$.
   1. By Exercise 2b, applied to the coarser chain of the levels t_k, a token is revealed in jump k with probability Δ_k, independently of the other positions. Show that two tokens are revealed in the same jump with probability $\rho = \sum_k \Delta_k^2$, and that, among all S increments that sum to one, equal increments $\Delta_k = 1/S$ minimize ρ.
   2. Show that with an exact network the sampler draws the two tokens of the quantized mixture from $(1-\rho)\,p + \rho\,p^1 \otimes p^2$.
   3. For a sequence of L tokens and equal increments, compute the expected number of pairs of tokens revealed in the same jump. How many steps keep it below one for $L = 1024$? Why does this count overstate the error of the sampler on text?

4. code **One token at a time.** An order-agnostic autoregressive sampler reveals exactly one token per step. Use `pair_model` with the linear `alpha_bar` of the quantized mixture and the mask index `K`.
   1. Write a sampler with two steps. The first reveals one of the two positions, chosen uniformly at random, from the prediction at the all-mask input with time input 1. The second reveals the other position from the prediction with time input $1/2$. Sample 20,000 pairs and compare their total variation distance to `table` with that of the thirty-two-step sampler.
   2. The same predictions define, for the order that reveals position 1 first, a table over all 1,024 pairs,

$$q_{12}(a, b) = p_\theta(\mathbf{x}_0^1 = a \mid [\mathrm{M}], [\mathrm{M}])\, p_\theta(\mathbf{x}_0^2 = b \mid a, [\mathrm{M}]), \tag{17.66}$$

   and the reverse order defines a table q_{21}. Compute both from `pair_model` with the time inputs of (a), and their total variation distances to `table` and to each other. An exact network makes the two tables equal. What does their difference say about the trained network?

5. **Two reverse parameterizations.** ◻ The reverse model Equation 17.63 averages the posterior over the network's prediction. Austin et al. (2021) instead set

$$p_\theta(\mathbf{x}_{t-1} \mid \mathbf{x}_t) \propto \sum_{\hat{\mathbf{x}}_0} q(\mathbf{x}_{t-1}, \mathbf{x}_t \mid \hat{\mathbf{x}}_0)\, \tilde{p}_\theta(\hat{\mathbf{x}}_0 \mid \mathbf{x}_t) \tag{17.67}$$

   for a network output $\tilde{p}_\theta$, which reweights each candidate by $q(\mathbf{x}_t \mid \hat{\mathbf{x}}_0)$. Let $q(\mathbf{x}_0 \mid \mathbf{x}_t)$ and $q(\mathbf{x}_{t-1} \mid \mathbf{x}_t)$ denote the conditionals of the forward process started from the data.

1. Take a single token. Show that if the prediction in Equation 17.63 equals $q(\mathbf{x}_0 \mid \mathbf{x}_t)$, the reverse model is the true reverse transition $q(\mathbf{x}_{t-1} \mid \mathbf{x}_t)$, for every kernel.
2. For the same token, show that the renormalized form yields $q(\mathbf{x}_{t-1} \mid \mathbf{x}_t)$ when $\tilde{p}_\theta(\cdot \mid \mathbf{x}_t)$ is the data distribution itself, and in general not when it is $q(\mathbf{x}_0 \mid \mathbf{x}_t)$. Under the uniform kernel, what happens to the evidence that $\mathbf{x}_t$ carries about $\mathbf{x}_0$ in the second case?
3. Show that under the absorbing kernel the weight $q(\mathbf{x}_t \mid \hat{\mathbf{x}}_0)$ is the same for every candidate at a masked position, so that the two forms agree.
4. For a sequence of L tokens, the renormalized form multiplies the network's output at position i by $q(\mathbf{x}_t^i \mid \hat{\mathbf{x}}_0^i)$, the evidence of the corrupted token at that position. Given (a) and (b), which positions of $\mathbf{x}_t$ does the network of each form have to read to produce the true reverse transition at position i?

6. code **Time-agnostic predictions.** ▯ This section argued that under the absorbing kernel the best prediction at a masked position does not depend on t (Ou et al. 2025).
   1. Write the argument as a short proof. Which property of the absorbing forward process does it use? Does the same hold for the uniform kernel?
   2. A step that reveals no token leaves the input of the next step unchanged. A sampler can therefore draw the reveal decisions first and evaluate a network without time input only in the jumps that reveal at least one token. Show that with S equally spaced levels under the linear schedule the expected number of evaluations is $S\,(1-(1-1/S)^L)$ for L tokens. Evaluate it for the pair with $S = 32$, for the binary images with $S = 100$, and for $L = S = 1024$.
   3. BERT (Devlin et al. 2018) is trained to predict masked tokens at a masking rate of 15%. By (a), its exact predictions are those that the sampler needs at mask patterns of that density. For sequences of $L = 512$ tokens, find the range of masked fractions that BERT sees in training, say within three standard deviations of 15%. What fraction of the steps of a run from the all-mask sequence, with $S = 512$ equally spaced levels under the linear schedule, shows the network a masked fraction in this range?
   4. `MaskedMLP` receives the time as an input. Fix the pattern in which the first token is masked and the second is observed, and compute the predictions of `pair_model` for all 32 values of the second token at $t \in \{1, 16, 31\}$, that is, at time inputs $t/32$. Then compute two kinds of Kullback–Leibler divergence, each averaged over the marginal of the second token: that of the predictions at $t = 16$ and at $t = 31$ from those at $t = 1$, and that of the predictions at each t from the exact conditionals of `table` (clamp probabilities at 10^{-12}). What do the sizes of the two kinds say about the use that the trained network makes of its time input?

7. code **Floor of an exact network.** ▯ Let a schedule with $\bar{\alpha}_0 = 1$ and $\bar{\alpha}_T = 0$ have the increments $\Delta_t = \bar{\alpha}_{t-1} - \bar{\alpha}_t$, and consider the two tokens of the quantized mixture.
   1. Show that the bound Equation 17.65 of an exact network, averaged over the true table, is

$$H(X^1, X^2) + I(X^1; X^2) \sum_{t=1}^{T} \Delta_t^2. \tag{17.68}$$

   Write the term of step t with the three mask patterns and their probabilities; a telescoping sum of $(1-\bar{\alpha}_t)^2$ evaluates the sum over t. Check that the linear

schedule gives $H(X^1, X^2) + I(X^1; X^2)/T$.

2. By Exercise 3a, $\sum_t \Delta_t^2$ is the probability that the T-step sampler reveals both tokens in the same step. Explain why the excess of the bound over the entropy is the mutual information times this probability.
3. Evaluate the sum for the schedule $\bar{\alpha}_t = 1-(t/T)^2$ in closed form and at $T = 32$, and compare the floor with that of the linear schedule. Why does the choice of schedule cease to matter as $T \to \infty$?
4. Write a function with the call signature of `MaskedMLP` that returns the logarithms of the exact conditionals computed from `table`, and of the marginals where both tokens are masked (clamp probabilities at 10^{-12}). Pass it to `exact_bound` with each of the two schedules as `alpha_bar`.
5. Train `MaskedMLP` with `masked_loss` under the schedule of (c), with the settings used for `pair_model`, and compare its exact bound with its floor and with the exact bound of `pair_model`.

8. **The order-agnostic bound.** ◻ Let a network ignore the time, as the exact network does by Exercise 6a, and write $\ell(i, U) = -\log p_\theta(\mathbf{x}_0^i \mid \mathbf{x}_0^U)$ for its loss at position i when exactly the positions in the set U are unmasked. The continuous-time limit of the bound in Section 17.8.2.3, written as an integral over the masking probability u, is then a weighted sum of the losses $\ell(i, U)$ over $i \notin U$. Ou et al. (2025) show that it equals the expected negative log-likelihood of an order-agnostic autoregressive model (Uria et al. 2014),

$$\mathbb{E}_\sigma\Big[\sum_{k=1}^{L} -\log p_\theta(\mathbf{x}_0^{\sigma(k)} \mid \mathbf{x}_0^{\sigma(1)}, \dots, \mathbf{x}_0^{\sigma(k-1)})\Big], \tag{17.69}$$

where σ is a uniformly random permutation of the L positions.

1. Find the weight of $\ell(i, U)$ in the integral when $|U| = L - m$. The Beta integral $\int_0^1 u^{m-1}(1-u)^{L-m}\,du = (m-1)!\,(L-m)!/L!$ evaluates it.
2. Find the probability that a random order reveals the positions of U first and position i next, and conclude that the two expressions agree.
3. Every order gives the same value for an exact network. Which value is it? Compare with Exercise 7a as $T \to \infty$.
4. Use Jensen's inequality to show that the expression also bounds the negative log-likelihood of the model that draws an order uniformly at random and then samples the tokens in that order. Which sampler of Exercise 4 implements this model for two tokens?

9. code **Confidence-ordered unmasking.** ◻ This section computed the output table of confidence remasking on the quantized mixture exactly.

1. Under the rule of Nie, Zhu, You, et al. (2025), both steps take the most probable value instead of a draw: in the first step the candidates are the modes of p^1 and p^2, the one with the higher probability is kept, and in the second step the other token is the mode of its conditional given the kept one. Which pair does this sampler output, and with which probability? Compute its total variation distance to `table`.
2. Why can the bound Equation 17.65 not detect the error of confidence remasking?

3. Repeat the comparison of the four orders on a coarser grid, $K = 8$, on which each mode covers only a few cells. Which of the two errors, that of the one-step draw and that of confidence remasking, depends on the grid, and why?

10. [extended] **A masked diffusion language model.** ▯▯ Prabhudesai et al. (2025) report that masked diffusion models make better use of repeated data than autoregressive models. This problem repeats the data on purpose. Use the character-level windows `d2l.TimeMachine(batch_size=64, num_steps=128, tokenization='char', num_train=100000, num_val=3000)` of Section 8.2, whose 100,000 training windows start at the first 100,000 positions of *The Time Machine*; 2,000 updates on minibatches of 64 windows of 128 characters then pass over these positions about 160 times.

    1. Train the `GPT` of Section 11.2 on these windows for 2,000 updates with its recipe, `LMOptimizer`, and record its training and validation losses every 200 updates. At which update does the validation loss reach its minimum, and what do the two losses do after it?
    2. Build the denoiser as a subclass of `d2l.GPT` with `vocab_size + 1` tokens, the last one the mask, and a forward pass that accepts the time and ignores it, as this section's argument allows. (Hint: to drop the causal mask, override the nested class `CausalAttention` and call the framework's fused attention without the mask.) Train it with `masked_loss` and the linear `alpha_bar` of $T = 128$ steps for 2,000 updates.
    3. Every 200 updates, estimate its bound in bits per character on the validation windows, with several masks per window. Plot it next to the validation loss of `GPT` in bits per character. Compare the best value of each curve and the update at which it is reached. Expect the denoiser to stay well behind: in a scoping run on the WikiText-103 bytes of Section 11.2, an untuned denoiser of this size reached about 2.3 nats per byte after 1,000 updates, where `GPT` reaches about 1.3. The gap is the cost that the scaling studies measure, not a bug.
    4. Continue training the denoiser to 8,000 updates if its bound still falls at update 2,000. Which model begins to overfit later? How does training on randomly masked windows change what the denoiser can memorize from a window it has seen many times?
    5. Sample windows with $S \in \{128, 32, 8\}$ steps and compare them with samples of `GPT`. Where do the few-step samples fail? Relate the failures to Exercise 3c.

17.9 Further Reading

The chapter taught one idea on problems small enough to check: a network that predicts the clean point from a noisy one at every noise level, trained by regression and turned into a sampler. Each entry states what a later work changes; nothing in the book depends on it. Distillation by distribution matching, adversarial post-training and the pitfalls of FID are on Section 16.8; Schrödinger bridges, optimal couplings and sliced score matching on Section 38.7.

17.9.1 Energy-Based Models and Score Matching

Training energy-based models. Song and Kingma (2021) survey the ways to fit a density known up to its normalizer. The likelihood gradient needs samples from the model, and the methods differ in where their chains start: contrastive divergence restarts them at the

data (Hinton 2002) and so cannot correct mode weights, persistent chains keep their states (Tieleman 2008), and Du and Mordatch (2019) scaled Langevin chains with a replay buffer to images. Short chains restarted from noise give a good sampler but a poor density (Nijkamp et al. 2019). Grathwohl et al. (2020) read a classifier's logits as the energy of inputs and labels, so one network classifies and generates.

Noise-contrastive estimation. Gutmann and Hyvärinen (2012) fit an unnormalized model by logistic regression of data against samples from a known noise distribution, the classifier of Section 16.1 with fixed noise in place of a generator.

What score matching cannot see. Wenliang and Kanagawa (2020) show that the score is blind to the weights of well-separated components, as Section 17.1 measures, and Koehler et al. (2022) tie the efficiency of score matching to isoperimetric constants, which are poor when modes are far apart.

Correcting and reusing Langevin steps. An accept-reject test removes the step-size bias of the Langevin update (Roberts and Tweedie 1996) but needs density ratios, which a score network does not give. Stochastic-gradient Langevin dynamics runs the update on the posterior of a network's weights with minibatch gradients (Welling and Teh 2011).

Empirical Bayes. Tweedie's formula comes from empirical Bayes, where a prior is estimated from the noisy observations it is applied to (Robbins 1956); Efron (2011) showed how the formula corrects selection bias, as in the largest of many noisy effect estimates.

17.9.2 Schedules, Weighting and Networks

Cosine schedule and learned variances. Nichol and Dhariwal (2021) destroy information more evenly across steps with a cosine schedule and learn each reverse step's variance from the bound, which improves likelihoods at fewer steps.

Choosing the training levels. Esser et al. (2024) find that a logit-normal distribution of training times, concentrated at intermediate noise, works best for rectified flow at every model size; later text-to-image models adopt it.

Weighting as multi-task learning. Hang et al. (2023) clamp the weight of each noise level at the minimum of its signal-to-noise ratio and a constant, one point in the weighting space of Section 17.5, reported to converge several times faster.

Zero terminal SNR. Against the leftover signal of Section 17.5, Lin, Liu, et al. (2024) rescale the schedule to end at zero signal, which needs a target other than the noise.

Resolution and averaging. At higher resolution the same noise level destroys less of an image, so Hoogeboom et al. (2023) shift the schedule toward more noise. Karras, Aittala, Lehtinen, et al. (2024) keep the magnitudes of activations and weights constant and store snapshots of two moving averages, from which an average of any half-life, fixed before training in Section 17.3, is reconstructed afterwards.

The U-Net at ImageNet scale. Dhariwal and Nichol (2021) ablated the U-Net of Ho et al. (2020) and kept multi-resolution attention, residual resampling and adaptive group normalization, in which the time and class embedding set a scale as well as a shift. The result, ADM, is the usual baseline.

17.9.3 Samplers

Exponential integrators. Dividing by α removes the part of the probability-flow ODE that is linear in $\mathbf{x}$. DPM-Solver (Lu, Zhou, et al. 2022) approximates the remaining integral of the predicted noise by a polynomial in $\log \bar{\sigma}$; its first-order version is DDIM (exercise 6). DPM-Solver++ (C. Lu et al. 2025) applies the idea to the clean-image prediction and stays stable at large guidance scales, and UniPC (W. Zhao et al. 2023) adds a corrector that gains an order of accuracy without an extra network call.

Noise inside a deterministic solver. Karras et al. (2022) raise the noise level slightly before each Heun step, adding a stochastic sampler's correction to a second-order method; it helps on some datasets and hurts on others. Restart sampling (Xu et al. 2023) alternates deterministic steps with one large jump back up the interval by forward noise, which contracts accumulated error.

Parallel sampling. Shih et al. (2023) refine all steps of a guessed trajectory at once by Picard iteration, trading compute for latency.

17.9.4 Guidance, Editing and Inverse Problems

Better guidance. Against the saturated colors of large guidance scales, dynamic thresholding (Saharia et al. 2022) clips the predicted clean image, and Sadat et al. (2025) down-weight the part of the update parallel to the conditional prediction.

Inversion and editing. The round trip of `ddim_invert` is inexact, more so under guidance. EDICT (Wallace et al. 2023) keeps two coupled copies of the state that invert each other exactly, and null-text inversion (Mokady et al. 2023) optimizes the unconditional text embedding so that the guided sampler retraces the inversion, which allows text-driven editing of real photographs.

Restoration without retraining. RePaint (Lugmayr et al. 2022) inpaints by re-imposing the noised known pixels at every step; DDRM (Kawar et al. 2022) solves linear inverse problems through the SVD of the measurement operator; DPS (Chung et al. 2023) follows the gradient of the measurement likelihood at the predicted clean image. ControlNet (L. Zhang et al. 2023) instead trains a copy of the encoder on edges, depth or pose.

17.9.5 Latent Spaces and Backbones

Latent systems. SDXL (Podell et al. 2024) scales the latent U-Net of Rombach et al. (2022) with two text encoders; SD3 (Esser et al. 2024) and FLUX.1 (Black Forest Labs 2024) are joint-attention transformers trained with rectified flow.

Better latents. REPA (S. Yu et al. 2025) aligns the transformer's hidden states with a pretrained image encoder and trains many times faster. DC-AE (Chen et al. 2025) compresses far more than eightfold with sharp reconstructions, and representation autoencoders (Zheng et al. 2026) use a frozen pretrained encoder such as DINO.

Transformers on pixels. Two lines of work drop the autoencoder that limits the latent models of Section 17.6. Hourglass diffusion transformers (Crowson et al. 2024) give the transformer the U-Net's multi-resolution shape, so its cost grows linearly with the pixels. JiT (Li and He 2026) keeps pixel patches of 16×16 or more and predicts the clean image, while the loss stays on the velocity.

17.9.6 Few-Step Generation

Distillation and consistency. Progressive distillation (Salimans and Ho 2022) repeatedly halves a teacher's steps; it introduced $\mathbf{v}$-prediction because a few-step student spends most of its evaluations at low signal-to-noise ratio, where a noise predictor's $\hat{\mathbf{x}}_0$ divides by a small α. Improved consistency training (Song and Dhariwal 2024) needs no teacher. sCM (Lu and Song 2025) trains consistency models in continuous time at ImageNet scale, which needs more than the identity of Section 17.7: a path that rotates between data and noise, a normalized tangent term and a warm-up.

General flow maps. Consistency trajectory models (Kim et al. 2024) jump between any two times; shortcut models (Frans et al. 2025) require one step of twice the size to equal two steps; improved MeanFlow (Geng et al. 2026) regresses on the instantaneous velocity and takes the guidance scale as an input. Boffi et al. (2025) turn any distillation scheme into training from scratch.

17.9.7 Likelihood

Bounds and weightings. Variational diffusion models (Kingma et al. 2021) learn the schedule to reduce the variance of the bound. Y. Song, Durkan, et al. (2021) find the weighting under which score matching bounds the likelihood, and Lu, Zheng, et al. (2022) and Zheng et al. (2023) train and evaluate the likelihood of the probability-flow ODE.

17.9.8 Evaluation, Memorization and Generalization

Text–image agreement. CLIPScore (Hessel et al. 2021) compares image and caption in a joint embedding; GenEval (Ghosh et al. 2023) checks counts, colors and positions with an object detector.

Memorization. Carlini et al. (2023) extract near-copies of training images from text-to-image models, and Somepalli et al. (2023) find replicated content by retrieval. Biroli et al. (2024) find two times in the reverse process: speciation, when a sample commits to a class, and collapse, when it commits to a training point.

Why networks generalize. The exact optimum of the regression on a finite set reproduces training points, and Bertrand et al. (2025) show that the noisy target is not what prevents this. Denoisers trained on disjoint halves of a large set learn nearly the same score (Kadkhodaie et al. 2024), and locality and equivariance alone predict their outputs (Kamb and Ganguli 2025). Massias and Bertrand (2026) survey the question.

17.9.9 Video

Video diffusion. Ho et al. (2022) factorize attention over space and time; Stable Video Diffusion (Blattmann et al. 2023) shows that curating video data matters as much as the architecture; Movie Gen (Polyak et al. 2024) trains one transformer on joint image and video latents.

17.9.10 Discrete Diffusion and Diffusion Language Models

Continuous time and ratios. In continuous time the masked chain of Section 17.8 becomes a continuous-time Markov chain (Campbell et al. 2022), and the score becomes ratios $p_t(\mathbf{y})/p_t(\mathbf{x})$ between sequences that differ in one position. Concrete score matching

(C. Meng, Choi, et al. 2022) and SEDD (Lou et al. 2024) learn these ratios; for the absorbing kernel a ratio is a time factor times the clean conditional, what the masked denoiser learns. Discrete flow matching (Gat et al. 2024; Campbell et al. 2024) builds the same models from probability paths, including joint discrete and continuous data such as protein sequences and structures.

The time input. Ou et al. (2025) show that the continuous-time bound equals the loss of an order-agnostic autoregressive model. K. Zheng, Chen, Mao, et al. (2025) show that sampling categories in low precision lowers the effective temperature and flatters reported generative perplexity.

Uniform noise and revision. Under a uniform kernel every token can change at every step: Schiff et al. (2025) add guidance to such models, and Sahoo et al. (2025) relate them to a Gaussian process and distill them to few steps. ReMDM (G. Wang et al. 2025) adds remasking to the masked sampler so that early mistakes can be revised. On puzzles such as Sudoku, a reveal order chosen by the model helps substantially (Kim et al. 2025), the other side of the mode-seeking measured in Section 17.8.

Scale. Masked diffusion follows autoregressive scaling laws with a constant-factor compute gap (Nie, Zhu, Du, et al. 2025) and overtakes autoregression under repeated data (Prabhudesai et al. 2025). LLaDA (Nie, Zhu, You, et al. 2025) trains an 8-billion-parameter model from scratch and decodes in blocks; Dream (J. Ye et al. 2025) and DiffuLLaMA (Gong et al. 2025) adapt autoregressive models instead. Block diffusion (Arriola et al. 2025) makes the blocks part of the model, interpolating between a GPT and a masked model, and Fast-dLLM (Wu et al. 2026) adds caching and parallel decoding without retraining.

Images and the debate. MaskGIT (Chang et al. 2022) unmasks the codes of an image tokenizer in eight to twelve confidence-ordered steps, and Muse (Chang et al. 2023) conditions it on text. Dieleman (2026a) traces why masked diffusion displaced continuous diffusion for language and why continuous models are returning.

17.9.11 Post-Training by Reward and Preference

Policy gradients. After pretraining, a reward can replace the regression target, with the reverse steps as actions. DDPO (Black et al. 2024) applies the clipped policy-gradient update of Section 15.8 to the Gaussian reverse steps, and Flow-GRPO (Jie Liu et al. 2025) brings the group-relative update to flow models through a stochastic sampler with the same marginals.

Preferences. Diffusion-DPO (Wallace et al. 2024) adapts direct preference optimization, built on the Bradley–Terry model of Section 15.3, by replacing the intractable log-likelihood with the variational bound of Section 17.5.

References

Abadi, Martín, Paul Barham, Jianmin Chen, et al. 2016. "TensorFlow: A System for Large-Scale Machine Learning." *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, 265–83. https://www.usenix.org/conference/osdi16/technical-sessions/presentation/abadi.

Abnar, Samira, Harshay Shah, Dan Busbridge, Alaaeldin Mohamed Elnouby Ali, Josh Susskind, and Vimal Thilak. 2025. "Parameters Vs FLOPs: Scaling Laws for Optimal Sparsity for Mixture-of-Experts Language Models." *arXiv Preprint arXiv:2501.12370*.

Achiam, Joshua. 2017. *Advanced Policy Gradient Methods*. Lecture, UC Berkeley CS 294-112: Deep Reinforcement Learning, https://rail.eecs.berkeley.edu/deeprlcourse-fa17/f17docs/lecture_13_advanced_pg.pdf.

Achiam, Joshua. 2018. *Spinning up in Deep RL*. Online, OpenAI; PyTorch rewrite Jan 2020, https://spinningup.openai.com/.

Agarwal, Alekh, Nan Jiang, Sham M Kakade, and Wen Sun. 2019. *Reinforcement Learning: Theory and Algorithms*. Working draft monograph, continuously revised, https://rltheorybook.github.io/.

Agarwal, Alekh, Sham M. Kakade, Jason D. Lee, and Gaurav Mahajan. 2021. "On the Theory of Policy Gradient Methods: Optimality, Approximation, and Distribution Shift." *Journal of Machine Learning Research* 22 (98): 1–76.

Agarwal, Rishabh, Max Schwarzer, Pablo Samuel Castro, Aaron Courville, and Marc G Bellemare. 2021. "Deep Reinforcement Learning at the Edge of the Statistical Precipice." *Advances in Neural Information Processing Systems* 34. https://arxiv.org/abs/2108.13264.

Agarwal, Rishabh, Nino Vieillard, Yongchao Zhou, et al. 2024. "On-Policy Distillation of Language Models: Learning from Self-Generated Mistakes." *International Conference on Learning Representations*. https://arxiv.org/abs/2306.13649.

Agrawal, Shipra, and Navin Goyal. 2012. "Analysis of Thompson Sampling for the Multi-Armed Bandit Problem." *Proceedings of the 25th Annual Conference on Learning Theory*, Proceedings of machine learning research, vol. 23: 39.1–26. https://proceedings.mlr.press/v23/agrawal12.html.

Ahmadian, Arash, Chris Cremer, Matthias Gallé, et al. 2024. "Back to Basics: Revisiting

REINFORCE-Style Optimization for Learning from Human Feedback in LLMs." *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics*. https://arxiv.org/abs/2402.14740.

Ahn, Kwangjun, Byron Xu, Natalie Abreu, et al. 2025. "Dion: Distributed Orthonormalized Updates." *arXiv Preprint arXiv:2504.05295*.

Ainslie, Joshua, James Lee-Thorp, Michiel de Jong, Yury Zemlyanskiy, Federico Lebrón, and Sumit Sanghai. 2023. "GQA: Training Generalized Multi-Query Transformer Models from Multi-Head Checkpoints." *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, 4895–901.

Ainsworth, Samuel K., Jonathan Hayase, and Siddhartha Srinivasa. 2023. "Git Re-Basin: Merging Models Modulo Permutation Symmetries." *International Conference on Learning Representations*.

Akyürek, Ekin, Dale Schuurmans, Jacob Andreas, Tengyu Ma, and Denny Zhou. 2023. "What Learning Algorithm Is in-Context Learning? Investigations with Linear Models." *International Conference on Learning Representations*. https://arxiv.org/abs/2211.15661.

Alayrac, Jean-Baptiste, Jeff Donahue, Pauline Luc, et al. 2022. "Flamingo: A Visual Language Model for Few-Shot Learning." *ArXiv:2204.14198*. https://arxiv.org/abs/2204.14198.

Albergo, Michael S., Nicholas M. Boffi, and Eric Vanden-Eijnden. 2023. "Stochastic Interpolants: A Unifying Framework for Flows and Diffusions." *arXiv Preprint arXiv:2303.08797*.

Amari, Shun-ichi. 1998. "Natural Gradient Works Efficiently in Learning." *Neural Computation* 10 (2): 251–76.

Ameisen, Emmanuel, Jack Lindsey, Adam Pearce, Wes Gurnee, Nicholas L. Turner, et al. 2025. *Circuit Tracing: Revealing Computational Graphs in Language Models*. Transformer Circuits Thread, https://transformer-circuits.pub/2025/attribution-graphs/methods.html.

Amsel, Noah, David Persson, Christopher Musco, and Robert M. Gower. 2026. "The Polar Express: Optimal Matrix Sign Methods and Their Application to the Muon Algorithm." *International Conference on Learning Representations*.

Amsel, Noah, Jack Zhang, Kwangjun Ahn, et al. 2026. "Dion3: Full-Stack Orthogonal Updates." *arXiv Preprint arXiv:2608.11612*.

Anderson, Brian D. O. 1982. "Reverse-Time Diffusion Equation Models." *Stochastic Processes and Their Applications* 12 (3): 313–26.

Andrychowicz, Marcin, Anton Raichuk, Piotr Stańczyk, et al. 2021. "What Matters for on-Policy Deep Actor-Critic Methods? A Large-Scale Study." *International Conference on Learning Representations*. https://arxiv.org/abs/2006.05990.

Ansel, Jason, Edward Yang, Horace He, et al. 2024. "PyTorch 2: Faster Machine Learning Through Dynamic Python Bytecode Transformation and Graph Compilation." *29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 929–47. https://doi.org/10.1145/3620665.3640366.

Arjovsky, Martin, and Léon Bottou. 2017. "Towards Principled Methods for Training Generative Adversarial Networks." *International Conference on Learning Representations*.

Arjovsky, Martin, Soumith Chintala, and Léon Bottou. 2017. "Wasserstein Generative Adversarial Networks." *Proceedings of the 34th International Conference on Machine Learning*, 214–23.

Arora, Sanjeev, Rong Ge, Yingyu Liang, Tengyu Ma, and Yi Zhang. 2017. "Generalization and Equilibrium in Generative Adversarial Nets (GANs)." *International Conference on Machine Learning*, 224–32.

Arora, Simran, Sabri Eyuboglu, Aman Timalsina, et al. 2024. "Zoology: Measuring and Improving Recall in Efficient Language Models." *International Conference on Learning Representations*. https://arxiv.org/abs/2312.04927.

Arora, Simran, Sabri Eyuboglu, Michael Zhang, et al. 2024. "Simple Linear Attention Language Models Balance the Recall-Throughput Tradeoff." *International Conference on Machine Learning*.

Arriola, Marianne, Aaron Gokaslan, Justin T. Chiu, et al. 2025. "Block Diffusion: Interpolating Between Autoregressive and Diffusion Language Models." *International Conference on Learning Representations*. https://arxiv.org/abs/2503.09573.

Auer, Peter, Nicolò Cesa-Bianchi, and Paul Fischer. 2002. "Finite-Time Analysis of the Multiarmed Bandit Problem." *Machine Learning* 47 (2-3): 235–56.

Austin, Jacob, Daniel D. Johnson, Jonathan Ho, Daniel Tarlow, and Rianne van den Berg. 2021. "Structured Denoising Diffusion Models in Discrete State-Spaces." *Advances in Neural Information Processing Systems* 34.

Ba, Jimmy Lei, Jamie Ryan Kiros, and Geoffrey E Hinton. 2016. "Layer Normalization." *ArXiv:1607.06450*. https://arxiv.org/abs/1607.06450.

Baevski, Alexei, and Michael Auli. 2018. "Adaptive Input Representations for Neural Language Modeling." *International Conference on Learning Representations*. https://openreview.net/forum?id=ByxZX20qFQ.

Bahdanau, Dzmitry, Kyunghyun Cho, and Yoshua Bengio. 2015. "Neural Machine Translation by Jointly Learning to Align and Translate." *International Conference on Learning Representations*. https://doi.org/10.48550/arXiv.1409.0473.

Baird, Leemon. 1995. "Residual Algorithms: Reinforcement Learning with Function Approximation." *International Conference on Machine Learning*, 30–37.

Bakker, Bram. 2001. "Reinforcement Learning with Long Short-Term Memory." *Advances*

in Neural Information Processing Systems 14.

Ball, Philip J., Laura Smith, Ilya Kostrikov, and Sergey Levine. 2023. "Efficient Online Reinforcement Learning with Offline Data." *International Conference on Machine Learning*, 1577–94.

Bao, Fan, Chongxuan Li, Jun Zhu, and Bo Zhang. 2022. "Analytic-DPM: An Analytic Estimate of the Optimal Reverse Variance in Diffusion Probabilistic Models." *International Conference on Learning Representations*. https://arxiv.org/abs/2201.06503.

Barbero, Federico, Álvaro Arroyo, Xiangming Gu, et al. 2025. "Why Do LLMs Attend to the First Token?" *Conference on Language Modeling (COLM)*. https://arxiv.org/abs/2504.02732.

Barbero, Federico, Alex Vitvitskyi, Christos Perivolaropoulos, Razvan Pascanu, and Petar Veličković. 2025. "Round and Round We Go! What Makes Rotary Positional Encodings Useful?" *International Conference on Learning Representations*. https://arxiv.org/abs/2410.06205.

Barto, Andrew G, Richard S Sutton, and Charles W Anderson. 1983. "Neuronlike Adaptive Elements That Can Solve Difficult Learning Control Problems." *IEEE Transactions on Systems, Man, and Cybernetics*, no. 5: 834–46.

Beck, Maximilian, Korbinian Pöppel, Markus Spanring, et al. 2024. "xLSTM: Extended Long Short-Term Memory." *Advances in Neural Information Processing Systems* 37. https://arxiv.org/abs/2405.04517.

Behrouz, Ali, Peilin Zhong, and Vahab Mirrokni. 2025. "Titans: Learning to Memorize at Test Time." *arXiv Preprint arXiv:2501.00663*.

Bellemare, Marc G, Will Dabney, and Rémi Munos. 2017. "A Distributional Perspective on Reinforcement Learning." *International Conference on Machine Learning*, 449–58.

Bellman, Richard. 1952. "On the Theory of Dynamic Programming." *Proceedings of the National Academy of Sciences* 38 (8): 716–19. https://doi.org/10.1073/pnas.38.8.716.

Bellman, Richard. 1957a. "A Markovian Decision Process." *Journal of Mathematics and Mechanics* 6 (5): 679–84. http://www.jstor.org/stable/24900506.

Bellman, Richard. 1957b. *Dynamic Programming*. Princeton University Press.

Beltagy, Iz, Matthew E Peters, and Arman Cohan. 2020. "Longformer: The Long-Document Transformer." *ArXiv:2004.05150*. https://arxiv.org/abs/2004.05150.

Bengio, Yoshua, Aaron Courville, and Pascal Vincent. 2013. "Representation Learning: A Review and New Perspectives." *IEEE Transactions on Pattern Analysis and Machine Intelligence* 35 (8): 1798–828.

Bergsma, Shane, Nolan Dey, Gurpreet Gosal, Gavia Gray, Daria Soboleva, and Joel Hestness. 2025a. "Power Lines: Scaling Laws for Weight Decay and Batch Size in LLM Pre-

Training." *Advances in Neural Information Processing Systems*. https://arxiv.org/abs/2505.13738.

Bergsma, Shane, Nolan Dey, Gurpreet Gosal, Gavia Gray, Daria Soboleva, and Joel Hestness. 2025b. "Straight to Zero: Why Linearly Decaying the Learning Rate to Zero Works Best for LLM Pretraining." *International Conference on Learning Representations*. https://arxiv.org/abs/2502.15938.

Bergstra, James, Olivier Breuleux, Frédéric Bastien, et al. 2010. "Theano: A CPU and GPU Math Compiler in Python." *Proc. 9th Python in Science Conference* 1: 3–10. https://www.iro.umontreal.ca/~lisa/pointeurs/theano_scipy2010.pdf.

Bernstein, Jeremy. 2025. *Deriving Muon*. Https://jeremybernste.in/writing/deriving-muon.

Bernstein, Jeremy, and Laker Newhouse. 2024. "Old Optimizer, New Norm: An Anatomy." *arXiv Preprint arXiv:2409.20325*.

Bertrand, Quentin, Anne Gagneux, Mathurin Massias, and Rémi Emonet. 2025. "On the Closed-Form of Flow Matching: Generalization Does Not Arise from Target Stochasticity." *Advances in Neural Information Processing Systems*. https://arxiv.org/abs/2506.03719.

Bertsekas, Dimitri P. 2025. *A Course in Reinforcement Learning*. 2nd ed. Athena Scientific.

Besiroglu, Tamay, Ege Erdil, Matthew Barnett, and Josh You. 2024. "Chinchilla Scaling: A Replication Attempt." *arXiv Preprint arXiv:2404.10102*.

Beyer, Lucas, Xiaohua Zhai, and Alexander Kolesnikov. 2022. "Better Plain ViT Baselines for ImageNet-1k." *arXiv Preprint arXiv:2205.01580*.

Bhatt, Aditya, Daniel Palenicek, Boris Belousov, et al. 2024. "CrossQ: Batch Normalization in Deep Reinforcement Learning for Greater Sample Efficiency and Simplicity." *International Conference on Learning Representations*.

Bick, Aviv, Kevin Y. Li, Eric P. Xing, J. Zico Kolter, and Albert Gu. 2024. "Transformers to SSMs: Distilling Quadratic Knowledge to Subquadratic Models." *Advances in Neural Information Processing Systems*.

Bińkowski, Mikołaj, Danica J. Sutherland, Michael Arbel, and Arthur Gretton. 2018. "Demystifying MMD GANs." *International Conference on Learning Representations*. https://arxiv.org/abs/1801.01401.

Biroli, Giulio, Tony Bonnaire, Valentin de Bortoli, and Marc Mézard. 2024. "Dynamical Regimes of Diffusion Models." *Nature Communications* 15: 9957. https://arxiv.org/abs/2402.18491.

Black Forest Labs. 2024. *FLUX.1 [Schnell] Model Card*. Https://huggingface.co/black-forest-labs/FLUX.1-schnell.

Black, Kevin, Michael Janner, Yilun Du, Ilya Kostrikov, and Sergey Levine. 2024. "Training Diffusion Models with Reinforcement Learning." *International Conference on Learning*

Representations. https://arxiv.org/abs/2305.13301.

Blattmann, Andreas, Tim Dockhorn, Sumith Kulal, et al. 2023. "Stable Video Diffusion: Scaling Latent Video Diffusion Models to Large Datasets." *arXiv Preprint arXiv:2311.15127*. https://arxiv.org/abs/2311.15127.

Blau, Yochai, and Tomer Michaeli. 2018. "The Perception-Distortion Tradeoff." *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 6228–37.

Blelloch, Guy E. 1990. *Prefix Sums and Their Applications*. CMU-CS-90-190. School of Computer Science, Carnegie Mellon University. https://www.cs.cmu.edu/~scandal/papers/CMU-CS-90-190.html.

Boffi, Nicholas M., Michael S. Albergo, and Eric Vanden-Eijnden. 2025. "How to Build a Consistency Model: Learning Flow Maps via Self-Distillation." *Advances in Neural Information Processing Systems*. https://arxiv.org/abs/2505.18825.

Boissin, Thibaut, Thomas Massena, Franck Mamalet, and Mathieu Serrurier. 2025. "Turbo-Muon: Almost-Orthogonal Pre-Conditioning for Fast Muon Updates." *arXiv Preprint arXiv:2512.04632*.

Bottou, Léon, Frank E. Curtis, and Jorge Nocedal. 2018. "Optimization Methods for Large-Scale Machine Learning." *SIAM Review* 60 (2): 223–311.

Boyd, Stephen, and Lieven Vandenberghe. 2004. *Convex Optimization*. Cambridge University Press. https://web.stanford.edu/~boyd/cvxbook/.

Bradley, Ralph Allan, and Milton E Terry. 1952. "Rank Analysis of Incomplete Block Designs: I. The Method of Paired Comparisons." *Biometrika* 39 (3/4): 324–45. https://doi.org/10.1093/biomet/39.3-4.324.

Brandfonbrener, David, Alberto Bietti, Jacob Buckman, Romain Laroche, and Joan Bruna. 2022. "When Does Return-Conditioned Supervised Learning Work for Offline Reinforcement Learning?" *Advances in Neural Information Processing Systems* 35.

Brock, Andrew, Jeff Donahue, and Karen Simonyan. 2019. "Large Scale GAN Training for High Fidelity Natural Image Synthesis." *International Conference on Learning Representations*. https://arxiv.org/abs/1809.11096.

Brown, Noam, and Tuomas Sandholm. 2017. "Libratus: The Superhuman AI for No-Limit Poker." *IJCAI*, 5226–28. https://doi.org/10.24963/ijcai.2017/772.

Brown, Tom, Benjamin Mann, Nick Ryder, et al. 2020. "Language Models Are Few-Shot Learners." *Advances in Neural Information Processing Systems* 33: 1877–901. https://arxiv.org/abs/2005.14165.

Bubeck, Sébastien, and Nicolò Cesa-Bianchi. 2012. "Regret Analysis of Stochastic and Nonstochastic Multi-Armed Bandit Problems." *Foundations and Trends in Machine Learning* 5 (1): 1–122. https://doi.org/10.1561/2200000024.

Burda, Yuri, Harrison Edwards, Amos Storkey, and Oleg Klimov. 2019. "Exploration by Random Network Distillation." *International Conference on Learning Representations*. https://arxiv.org/abs/1810.12894.

Cai, Tianle, Yuhong Li, Zhengyang Geng, et al. 2024. "Medusa: Simple LLM Inference Acceleration Framework with Multiple Decoding Heads." *International Conference on Machine Learning*, 5209–35. https://arxiv.org/abs/2401.10774.

Campbell, Andrew, Joe Benton, Valentin De Bortoli, Tom Rainforth, George Deligiannidis, and Arnaud Doucet. 2022. "A Continuous Time Framework for Discrete Denoising Models." *Advances in Neural Information Processing Systems* 35. https://arxiv.org/abs/2205.14987.

Campbell, Andrew, Jason Yim, Regina Barzilay, Tom Rainforth, and Tommi Jaakkola. 2024. "Generative Flows on Discrete State-Spaces: Enabling Multimodal Flows with Applications to Protein Co-Design." *International Conference on Machine Learning*.

Carion, Nicolas, Francisco Massa, Gabriel Synnaeve, Nicolas Usunier, Alexander Kirillov, and Sergey Zagoruyko. 2020. "End-to-End Object Detection with Transformers." *European Conference on Computer Vision*, 213–29.

Carlini, Nicholas, Jamie Hayes, Milad Nasr, et al. 2023. "Extracting Training Data from Diffusion Models." *32nd USENIX Security Symposium*, 5253–70. https://arxiv.org/abs/2301.13188.

Cesista, Franz Louis, Jiacheng You, and Keller Jordan. 2025. *Squeezing 1–2% Efficiency Gains Out of Muon by Optimizing the Newton–Schulz Coefficients*. Http://leloykun.github.io/ponder/muon-opt-coeffs/.

Chang, Chi-Chih, Wei-Cheng Lin, Chien-Yu Lin, et al. 2025. "Palu: KV-Cache Compression with Low-Rank Projection." *International Conference on Learning Representations*. https://arxiv.org/abs/2407.21118.

Chang, Huiwen, Han Zhang, Jarred Barber, et al. 2023. "Muse: Text-to-Image Generation via Masked Generative Transformers." *Proceedings of the 40th International Conference on Machine Learning*, Proceedings of machine learning research, vol. 202: 4055–75.

Chang, Huiwen, Han Zhang, Lu Jiang, Ce Liu, and William T. Freeman. 2022. "MaskGIT: Masked Generative Image Transformer." *IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 11315–25. https://arxiv.org/abs/2202.04200.

Chen, Charlie, Sebastian Borgeaud, Geoffrey Irving, Jean-Baptiste Lespiau, Laurent Sifre, and John Jumper. 2023. "Accelerating Large Language Model Decoding with Speculative Sampling." *arXiv Preprint arXiv:2302.01318*.

Chen, Junyu, Han Cai, Junsong Chen, et al. 2025. "Deep Compression Autoencoder for Efficient High-Resolution Diffusion Models." *International Conference on Learning Representations*. https://arxiv.org/abs/2410.10733.

Chen, Lili, Kevin Lu, Aravind Rajeswaran, et al. 2021. "Decision Transformer: Reinforce-

ment Learning via Sequence Modeling." *Advances in Neural Information Processing Systems* 34.

Chen, Mark, Jerry Tworek, Heewoo Jun, Qiming Yuan, et al. 2021. "Evaluating Large Language Models Trained on Code." *arXiv Preprint arXiv:2107.03374*.

Chen, Shouyuan, Sherman Wong, Liangjian Chen, and Yuandong Tian. 2023. "Extending Context Window of Large Language Models via Positional Interpolation." *arXiv Preprint arXiv:2306.15595*.

Chen, Tianqi, Bing Xu, Chiyuan Zhang, and Carlos Guestrin. 2016. "Training Deep Nets with Sublinear Memory Cost." *arXiv Preprint arXiv:1604.06174*.

Chen, Xiangning, Chen Liang, Da Huang, Esteban Real, et al. 2023. "Symbolic Discovery of Optimization Algorithms." *Advances in Neural Information Processing Systems* 36. https://arxiv.org/abs/2302.06675.

Chen, Xinyue, Che Wang, Zijian Zhou, and Keith Ross. 2021. "Randomized Ensembled Double Q-Learning: Learning Fast Without a Model." *International Conference on Learning Representations*. https://arxiv.org/abs/2101.05982.

Chi, Cheng, Siyuan Feng, Yilun Du, et al. 2023. "Diffusion Policy: Visuomotor Policy Learning via Action Diffusion." *Robotics: Science and Systems*. https://arxiv.org/abs/2303.04137.

Chi, Ta-Chung, Ting-Han Fan, Alexander I. Rudnicky, and Peter J. Ramadge. 2023. "Dissecting Transformer Length Extrapolation via the Lens of Receptive Field Analysis." *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics*. https://arxiv.org/abs/2212.10356.

Child, Rewon, Scott Gray, Alec Radford, and Ilya Sutskever. 2019. "Generating Long Sequences with Sparse Transformers." *ArXiv:1904.10509*. https://arxiv.org/abs/1904.10509.

Cho, Kyunghyun, Bart Van Merriënboer, Dzmitry Bahdanau, and Yoshua Bengio. 2014. "On the Properties of Neural Machine Translation: Encoder–Decoder Approaches." *ArXiv:1409.1259*. https://arxiv.org/abs/1409.1259.

Chong, Min Jin, and David Forsyth. 2020. "Effectively Unbiased FID and Inception Score and Where to Find Them." *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 6070–79. https://arxiv.org/abs/1911.07023.

Choromanski, Krzysztof, Valerii Likhosherstov, David Dohan, et al. 2021. "Rethinking Attention with Performers." *International Conference on Learning Representations*.

Chowdhery, Aakanksha, Sharan Narang, Jacob Devlin, et al. 2022. "PaLM: Scaling Language Modeling with Pathways." *ArXiv:2204.02311*. https://arxiv.org/abs/2204.02311.

Christiano, Paul, Jan Leike, Tom B Brown, Miljan Martic, Shane Legg, and Dario Amodei. 2017. "Deep Reinforcement Learning from Human Preferences." *Advances in Neural Information Processing Systems* 30.

Christodoulou, Petros. 2019. "Soft Actor-Critic for Discrete Action Settings." *arXiv Preprint arXiv:1910.07207*.

Chu, Casey, Andrey Zhmoginov, and Mark Sandler. 2017. "CycleGAN, a Master of Steganography." *arXiv Preprint arXiv:1712.02950*. https://arxiv.org/abs/1712.02950.

Chua, Kurtland, Roberto Calandra, Rowan McAllister, and Sergey Levine. 2018. "Deep Reinforcement Learning in a Handful of Trials Using Probabilistic Dynamics Models." *Advances in Neural Information Processing Systems* 31.

Chung, Hyungjin, Jeongsol Kim, Michael T. Mccann, Marc L. Klasky, and Jong Chul Ye. 2023. "Diffusion Posterior Sampling for General Noisy Inverse Problems." *International Conference on Learning Representations*. https://arxiv.org/abs/2209.14687.

Chung, Junyoung, Caglar Gulcehre, KyungHyun Cho, and Yoshua Bengio. 2014. "Empirical Evaluation of Gated Recurrent Neural Networks on Sequence Modeling." *ArXiv:1412.3555*. https://arxiv.org/abs/1412.3555.

Clark, Aidan, Diego de las Casas, Aurelia Guy, et al. 2022. "Unified Scaling Laws for Routed Language Models." *International Conference on Machine Learning*, 4057–86. https://arxiv.org/abs/2202.01169.

Cohen, Jeremy M., Simran Kaur, Yuanzhi Li, J. Zico Kolter, and Ameet Talwalkar. 2021. "Gradient Descent on Neural Networks Typically Occurs at the Edge of Stability." *International Conference on Learning Representations*.

Conmy, Arthur, Augustine N. Mavor-Parker, Aengus Lynch, Stefan Heimersheim, and Adrià Garriga-Alonso. 2023. "Towards Automated Circuit Discovery for Mechanistic Interpretability." *Advances in Neural Information Processing Systems* 36. https://arxiv.org/abs/2304.14997.

Cordonnier, Jean-Baptiste, Andreas Loukas, and Martin Jaggi. 2020. "On the Relationship Between Self-Attention and Convolutional Layers." *International Conference on Learning Representations*. https://openreview.net/forum?id=HJlnC1rKPB.

Crowson, Katherine, Stefan Andreas Baumann, Alex Birch, Tanishq Mathew Abraham, Daniel Z. Kaplan, and Enrico Shippole. 2024. "Scalable High-Resolution Pixel-Space Image Synthesis with Hourglass Diffusion Transformers." *Proceedings of the 41st International Conference on Machine Learning*, Proceedings of machine learning research, vol. 235: 9550–75.

D'Angelo, Francesco, Maksym Andriushchenko, Aditya Varre, and Nicolas Flammarion. 2024. "Why Do We Need Weight Decay in Modern Deep Learning?" *Advances in Neural Information Processing Systems* 37. https://arxiv.org/abs/2310.04415.

Dahl, George E., Frank Schneider, Zachary Nado, et al. 2023. "Benchmarking Neural Network Training Algorithms." *arXiv Preprint arXiv:2306.07179*.

Dai, Damai, Chengqi Deng, Chenggang Zhao, et al. 2024. "DeepSeekMoE: Towards Ultimate Expert Specialization in Mixture-of-Experts Language Models." *arXiv Preprint*

arXiv:2401.06066.

Damian, Alex, Eshaan Nichani, and Jason D. Lee. 2023. "Self-Stabilization: The Implicit Bias of Gradient Descent at the Edge of Stability." *International Conference on Learning Representations*.

Dao, Tri. 2024. "FlashAttention-2: Faster Attention with Better Parallelism and Work Partitioning." *International Conference on Learning Representations*. https://arxiv.org/abs/2307.08691.

Dao, Tri, Daniel Y. Fu, Stefano Ermon, Atri Rudra, and Christopher Ré. 2022. "FlashAttention: Fast and Memory-Efficient Exact Attention with IO-Awareness." *Advances in Neural Information Processing Systems*.

Dao, Tri, and Albert Gu. 2024. "Transformers Are SSMs: Generalized Models and Efficient Algorithms Through Structured State Space Duality." *International Conference on Machine Learning*, 10041–71. https://arxiv.org/abs/2405.21060.

Darcet, Timothée, Maxime Oquab, Julien Mairal, and Piotr Bojanowski. 2024. "Vision Transformers Need Registers." *International Conference on Learning Representations*. https://arxiv.org/abs/2309.16588.

Dauphin, Yann N., Razvan Pascanu, Caglar Gulcehre, Kyunghyun Cho, Surya Ganguli, and Yoshua Bengio. 2014. "Identifying and Attacking the Saddle Point Problem in High-Dimensional Non-Convex Optimization." *Advances in Neural Information Processing Systems* 27: 2933–41.

De, Soham, Samuel L. Smith, Anushan Fernando, et al. 2024. "Griffin: Mixing Gated Linear Recurrences with Local Attention for Efficient Language Models." *arXiv Preprint arXiv:2402.19427*.

Dean, Jeffrey, Greg S Corrado, Rajat Monga, et al. 2012. "Large Scale Distributed Deep Networks." *Proceedings of the 25th International Conference on Neural Information Processing Systems, Volume 1*, 1223–31. https://doi.org/10.5555/2999134.2999271.

DeepSeek-AI. 2024. "DeepSeek-V2: A Strong, Economical, and Efficient Mixture-of-Experts Language Model." *arXiv Preprint arXiv:2405.04434*.

DeepSeek-AI. 2025. "DeepSeek-R1: Incentivizing Reasoning Capability in LLMs via Reinforcement Learning." *Nature*, ahead of print. https://doi.org/10.1038/s41586-025-09422-z.

DeepSeek-AI, Xiao Bi, Deli Chen, Guanting Chen, et al. 2024. "DeepSeek LLM: Scaling Open-Source Language Models with Longtermism." *arXiv Preprint arXiv:2401.02954*.

DeepSeek-AI, Aixin Liu, Bei Feng, Bing Xue, et al. 2024. "DeepSeek-V3 Technical Report." *arXiv Preprint arXiv:2412.19437*.

DeepSeek-AI, Aixin Liu, Aoxue Mei, Bangcai Lin, et al. 2025. "DeepSeek-V3.2: Pushing the Frontier of Open Large Language Models." *arXiv Preprint arXiv:2512.02556*.

Defazio, Aaron, Ashok Cutkosky, Harsh Mehta, and Konstantin Mishchenko. 2023. "Optimal Linear Decay Learning Rate Schedules and Further Refinements." *arXiv Preprint arXiv:2310.07831*.

Defazio, Aaron, Xingyu Alice Yang, Harsh Mehta, Konstantin Mishchenko, Ahmed Khaled, and Ashok Cutkosky. 2024. "The Road Less Scheduled." *Advances in Neural Information Processing Systems* 37. https://arxiv.org/abs/2405.15682.

Dehghani, Mostafa, Josip Djolonga, Basil Mustafa, et al. 2023. "Scaling Vision Transformers to 22 Billion Parameters." *International Conference on Machine Learning*, 7480–512.

Deshpande, Ishan, Ziyu Zhang, and Alexander G. Schwing. 2018. "Generative Modeling Using the Sliced Wasserstein Distance." *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*.

Dettmers, Tim, Mike Lewis, Sam Shleifer, and Luke Zettlemoyer. 2022. "8-Bit Optimizers via Block-Wise Quantization." *International Conference on Learning Representations*. https://arxiv.org/abs/2110.02861.

Devlin, Jacob, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2018. "BERT: Pre-Training of Deep Bidirectional Transformers for Language Understanding." *ArXiv:1810.04805*. https://arxiv.org/abs/1810.04805.

DeVries, Terrance, Adriana Romero, Luis Pineda, Graham W. Taylor, and Michal Drozdzal. 2019. "On the Evaluation of Conditional GANs." *arXiv Preprint arXiv:1907.08175*. https://arxiv.org/abs/1907.08175.

Dey, Nolan, Quentin Anthony, and Joel Hestness. 2024. *The Practitioner's Guide to the Maximal Update Parameterization*. Https://blog.eleuther.ai/mutransfer/.

Dey, Nolan, Gurpreet Gosal, Zhiming Chen, et al. 2023. "Cerebras-GPT: Open Compute-Optimal Language Models Trained on the Cerebras Wafer-Scale Cluster." *arXiv Preprint arXiv:2304.03208*.

Dhariwal, Prafulla, and Alex Nichol. 2021. "Diffusion Models Beat GANs on Image Synthesis." *Advances in Neural Information Processing Systems* 34.

Dieleman, Sander. 2024. *Diffusion Is Spectral Autoregression*. Blog post, 2 September 2024, https://sander.ai/2024/09/02/spectral-autoregression.html.

Dieleman, Sander. 2025. *Generative Modelling in Latent Space*. Blog post, 15 April 2025, https://sander.ai/2025/04/15/latents.html.

Dieleman, Sander. 2026a. *Continuous Diffusion Language Models*. Blog post, 24 August 2026, https://sander.ai/2026/08/24/continuous-dlms.html.

Dieleman, Sander. 2026b. *Learning the Integral of a Diffusion Model*. Blog post, 6 May 2026, https://sander.ai/2026/05/06/flow-maps.html.

Ding, Yiran, Li Lyna Zhang, Chengruidong Zhang, et al. 2024. "LongRoPE: Extending LLM

Context Window Beyond 2 Million Tokens." *International Conference on Machine Learning*, Proceedings of machine learning research, vol. 235: 11091–104. https://arxiv.org/abs/2402.13753.

Dinh, Laurent, Razvan Pascanu, Samy Bengio, and Yoshua Bengio. 2017. "Sharp Minima Can Generalize for Deep Nets." *Proceedings of the 34th International Conference on Machine Learning*, Proceedings of machine learning research, vol. 70: 1019–28. https://proceedings.mlr.press/v70/dinh17b.html.

Dong, Juechu, Boyuan Feng, Driss Guessous, Yanbo Liang, and Horace He. 2025. "Flex Attention: A Programming Model for Generating Optimized Attention Kernels." *Proceedings of Machine Learning and Systems (MLSys)*. https://arxiv.org/abs/2412.05496.

Dong, Yihe, Jean-Baptiste Cordonnier, and Andreas Loukas. 2021. "Attention Is Not All You Need: Pure Attention Loses Rank Doubly Exponentially with Depth." *International Conference on Machine Learning*, 2793–803.

Dosovitskiy, Alexey, Lucas Beyer, Alexander Kolesnikov, et al. 2021. "An Image Is Worth 16 x 16 Words: Transformers for Image Recognition at Scale." *International Conference on Learning Representations*. https://openreview.net/forum?id=YicbFdNTTy.

Dowson, D. C., and B. V. Landau. 1982. "The Fréchet Distance Between Multivariate Normal Distributions." *Journal of Multivariate Analysis* 12 (3): 450–55.

Draxler, Felix, Kambis Veschgini, Manfred Salmhofer, and Fred Hamprecht. 2018. "Essentially No Barriers in Neural Network Energy Landscape." *Proceedings of the 35th International Conference on Machine Learning*, Proceedings of machine learning research, vol. 80: 1309–18.

Du, Yilun, and Igor Mordatch. 2019. "Implicit Generation and Modeling with Energy Based Models." *Advances in Neural Information Processing Systems* 32. https://arxiv.org/abs/1903.08689.

Duchi, John, Elad Hazan, and Yoram Singer. 2011. "Adaptive Subgradient Methods for Online Learning and Stochastic Optimization." *Journal of Machine Learning Research* 12: 2121–59. https://jmlr.org/papers/v12/duchi11a.html.

Efron, Bradley. 2011. "Tweedie's Formula and Selection Bias." *Journal of the American Statistical Association* 106 (496): 1602–14.

Elhage, Nelson, Tristan Hume, Catherine Olsson, et al. 2022. "Toy Models of Superposition." *Transformer Circuits Thread*.

Elhage, Nelson, Neel Nanda, Catherine Olsson, et al. 2021. "A Mathematical Framework for Transformer Circuits." *Transformer Circuits Thread*.

Elsayed, Mohamed, Gautham Vasan, and A Rupam Mahmood. 2024. "Streaming Deep Reinforcement Learning Finally Works." *arXiv Preprint arXiv:2410.14606*.

Emmons, Scott, Benjamin Eysenbach, Ilya Kostrikov, and Sergey Levine. 2022. "RvS: What

Is Essential for Offline RL via Supervised Learning?" *International Conference on Learning Representations*.

Engstrom, Logan, Andrew Ilyas, Shibani Santurkar, et al. 2020. "Implementation Matters in Deep RL: A Case Study on PPO and TRPO." *International Conference on Learning Representations*.

Entezari, Rahim, Hanie Sedghi, Olga Saukh, and Behnam Neyshabur. 2022. "The Role of Permutation Invariance in Linear Mode Connectivity of Neural Networks." *International Conference on Learning Representations*.

Espeholt, Lasse, Hubert Soyer, Rémi Munos, et al. 2018. "IMPALA: Scalable Distributed Deep-RL with Importance Weighted Actor-Learner Architectures." *International Conference on Machine Learning*, 1407–16.

Esser, Patrick, Sumith Kulal, Andreas Blattmann, et al. 2024. "Scaling Rectified Flow Transformers for High-Resolution Image Synthesis." *Proceedings of the 41st International Conference on Machine Learning*, Proceedings of machine learning research, vol. 235: 12606–33. https://arxiv.org/abs/2403.03206.

Esser, Patrick, Robin Rombach, and Björn Ommer. 2021. "Taming Transformers for High-Resolution Image Synthesis." *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*.

Even-Dar, Eyal, and Yishay Mansour. 2003. "Learning Rates for Q-Learning." *Journal of Machine Learning Research* 5: 1–25.

Fan, Angela, Mike Lewis, and Yann Dauphin. 2018. "Hierarchical Neural Story Generation." *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics*, 889–98.

Fedus, William, Barret Zoph, and Noam Shazeer. 2022. "Switch Transformers: Scaling to Trillion Parameter Models with Simple and Efficient Sparsity." *Journal of Machine Learning Research* 23 (120): 1–39. https://arxiv.org/abs/2101.03961.

Feng, Leo, Frederick Tung, Mohamed Osama Ahmed, Yoshua Bengio, and Hossein Hajimirsadeghi. 2024. "Were RNNs All We Needed?" *arXiv Preprint arXiv:2410.01201*.

Foret, Pierre, Ariel Kleiner, Hossein Mobahi, and Behnam Neyshabur. 2021. "Sharpness-Aware Minimization for Efficiently Improving Generalization." *International Conference on Learning Representations*. https://arxiv.org/abs/2010.01412.

Frans, Kevin, Danijar Hafner, Sergey Levine, and Pieter Abbeel. 2025. "One Step Diffusion via Shortcut Models." *International Conference on Learning Representations*. https://arxiv.org/abs/2410.12557.

Frostig, Roy, Matthew James Johnson, and Chris Leary. 2018. "Compiling Machine Learning Programs via High-Level Tracing." In *Proceedings of Systems for Machine Learning*. https://mlsys.org/Conferences/2018/doc/19.pdf.

Fu, Daniel Y., Tri Dao, Khaled K. Saab, Armin W. Thomas, Atri Rudra, and Christopher Ré. 2023. "Hungry Hungry Hippos: Towards Language Modeling with State Space Models." *International Conference on Learning Representations*. https://arxiv.org/abs/2212.14052.

Fujimoto, Scott, and Shixiang Shane Gu. 2021. "A Minimalist Approach to Offline Reinforcement Learning." *Advances in Neural Information Processing Systems* 34.

Fujimoto, Scott, Herke van Hoof, and David Meger. 2018. "Addressing Function Approximation Error in Actor-Critic Methods." *International Conference on Machine Learning*, 1587–96.

Fujimoto, Scott, David Meger, and Doina Precup. 2019. "Off-Policy Deep Reinforcement Learning Without Exploration." *International Conference on Machine Learning*, 2052–62.

Gallici, Matteo, Mattie Fellows, Benjamin Ellis, et al. 2025. "Simplifying Deep Temporal Difference Learning." *International Conference on Learning Representations*. https://arxiv.org/abs/2407.04811.

Gao, Leo, John Schulman, and Jacob Hilton. 2023. "Scaling Laws for Reward Model Overoptimization." *International Conference on Machine Learning*. https://arxiv.org/abs/2210.10760.

Gao, Ruiqi, Emiel Hoogeboom, Jonathan Heek, Valentin De Bortoli, Kevin P. Murphy, and Tim Salimans. 2025. *Diffusion Meets Flow Matching: Two Sides of the Same Coin*. ICLR Blog Track, https://diffusionflow.github.io/.

Garg, Shivam, Dimitris Tsipras, Percy Liang, and Gregory Valiant. 2022. "What Can Transformers Learn in-Context? A Case Study of Simple Function Classes." *Advances in Neural Information Processing Systems* 35. https://arxiv.org/abs/2208.01066.

Garipov, Timur, Pavel Izmailov, Dmitrii Podoprikhin, Dmitry P. Vetrov, and Andrew Gordon Wilson. 2018. "Loss Surfaces, Mode Connectivity, and Fast Ensembling of DNNs." *Advances in Neural Information Processing Systems* 31: 8789–98.

Gat, Itai, Tal Remez, Neta Shaul, et al. 2024. "Discrete Flow Matching." *Advances in Neural Information Processing Systems* 37. https://arxiv.org/abs/2407.15595.

Gemma Team. 2024. "Gemma 2: Improving Open Language Models at a Practical Size." *arXiv Preprint arXiv:2408.00118*.

Gemma Team. 2025. "Gemma 3 Technical Report." *arXiv Preprint arXiv:2503.19786*.

Genevay, Aude, Gabriel Peyré, and Marco Cuturi. 2018. "Learning Generative Models with Sinkhorn Divergences." *Proceedings of the 21st International Conference on Artificial Intelligence and Statistics*.

Geng, Zhengyang, Mingyang Deng, Xingjian Bai, J. Zico Kolter, and Kaiming He. 2025. "Mean Flows for One-Step Generative Modeling." *arXiv Preprint arXiv:2505.13447*.

Geng, Zhengyang, Yiyang Lu, Zongze Wu, Eli Shechtman, J. Zico Kolter, and Kaiming He.

2026. "Improved Mean Flows: On the Challenges of Fastforward Generative Models." *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*. https://arxiv.org/abs/2512.02012.

Gers, Felix A., Jürgen Schmidhuber, and Fred Cummins. 2000. "Learning to Forget: Continual Prediction with LSTM." *Neural Computation* 12 (10): 2451–71. https://doi.org/10.1162/089976600300015015.

Geva, Mor, Roei Schuster, Jonathan Berant, and Omer Levy. 2021. "Transformer Feed-Forward Layers Are Key-Value Memories." *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*, 5484–95. https://arxiv.org/abs/2012.14913.

Ghadimi, Saeed, and Guanghui Lan. 2013. "Stochastic First- and Zeroth-Order Methods for Nonconvex Stochastic Programming." *SIAM Journal on Optimization* 23 (4): 2341–68. https://arxiv.org/abs/1309.5549.

Ghazvininejad, Marjan, Omer Levy, Yinhan Liu, and Luke Zettlemoyer. 2019. "Mask-Predict: Parallel Decoding of Conditional Masked Language Models." *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing*, 6112–21.

Gholami, Amir, Zhewei Yao, Sehoon Kim, Coleman Hooper, Michael W. Mahoney, and Kurt Keutzer. 2024. "AI and Memory Wall." *IEEE Micro* 44 (3): 33–39. https://doi.org/10.1109/MM.2024.3373763.

Ghorbani, Behrooz, Shankar Krishnan, and Ying Xiao. 2019. "An Investigation into Neural Net Optimization via Hessian Eigenvalue Density." *Proceedings of the 36th International Conference on Machine Learning*, Proceedings of machine learning research, vol. 97: 2232–41.

Ghosh, Dhruba, Hanna Hajishirzi, and Ludwig Schmidt. 2023. "GenEval: An Object-Focused Framework for Evaluating Text-to-Image Alignment." *Advances in Neural Information Processing Systems*. https://arxiv.org/abs/2310.11513.

Givens, Clark R., and Rae Michael Shortt. 1984. "A Class of Wasserstein Metrics for Probability Distributions." *Michigan Mathematical Journal* 31 (2): 231–40.

Gloeckle, Fabian, Badr Youbi Idrissi, Baptiste Rozière, David Lopez-Paz, and Gabriel Synnaeve. 2024. "Better & Faster Large Language Models via Multi-Token Prediction." *International Conference on Machine Learning*, 15706–34. https://arxiv.org/abs/2404.19737.

Glorioso, Paolo, Quentin Anthony, Yury Tokpanov, et al. 2024. "Zamba: A Compact 7B SSM Hybrid Model." *arXiv Preprint arXiv:2405.16712*.

Godbole, Varun, George E. Dahl, Justin Gilmer, Christopher J. Shallue, and Zachary Nado. 2023. *Deep Learning Tuning Playbook*. Https://github.com/google-research/tuning_playbook.

Goel, Karan, Albert Gu, Chris Donahue, and Christopher Ré. 2022. "It's Raw! Audio Generation with State-Space Models." *International Conference on Machine Learning*. https:

//arxiv.org/abs/2202.09729.

Gong, Mingming, Yanwu Xu, Chunyuan Li, Kun Zhang, and Kayhan Batmanghelich. 2019. "Twin Auxiliary Classifiers GAN." *Advances in Neural Information Processing Systems* 32. https://arxiv.org/abs/1907.02690.

Gong, Shansan, Shivam Agarwal, Yizhe Zhang, et al. 2025. "Scaling Diffusion Language Models via Adaptation from Autoregressive Models." *International Conference on Learning Representations*.

Goodfellow, Ian. 2016. "NIPS 2016 Tutorial: Generative Adversarial Networks." *arXiv Preprint arXiv:1701.00160*. https://arxiv.org/abs/1701.00160.

Goodfellow, Ian, Jean Pouget-Abadie, Mehdi Mirza, et al. 2014. "Generative Adversarial Nets." *Advances in Neural Information Processing Systems*, 2672–80. https://doi.org/10.5555/2969033.2969125.

Gotmare, Akhilesh, Nitish Shirish Keskar, Caiming Xiong, and Richard Socher. 2018. "A Closer Look at Deep Learning Heuristics: Learning Rate Restarts, Warmup and Distillation." *ArXiv:1810.13243*. https://arxiv.org/abs/1810.13243.

Goyal, Priya, Piotr Dollár, Ross Girshick, et al. 2017. "Accurate, Large Minibatch SGD: Training ImageNet in 1 Hour." *ArXiv:1706.02677*. https://arxiv.org/abs/1706.02677.

Grathwohl, Will, Ricky T. Q. Chen, Jesse Bettencourt, Ilya Sutskever, and David Duvenaud. 2019. "FFJORD: Free-Form Continuous Dynamics for Scalable Reversible Generative Models." *International Conference on Learning Representations*.

Grathwohl, Will, Kuan-Chieh Wang, Jörn-Henrik Jacobsen, David Duvenaud, Mohammad Norouzi, and Kevin Swersky. 2020. "Your Classifier Is Secretly an Energy Based Model and You Should Treat It Like One." *International Conference on Learning Representations*. https://arxiv.org/abs/1912.03263.

Grattafiori, Aaron, Abhimanyu Dubey, Abhinav Jauhri, et al. 2024. "The Llama 3 Herd of Models." *arXiv Preprint arXiv:2407.21783*.

Graves, Alex. 2013. "Generating Sequences with Recurrent Neural Networks." *ArXiv:1308.0850*. https://arxiv.org/abs/1308.0850.

Grazzi, Riccardo, Julien Siems, Jörg K. H. Franke, Arber Zela, Frank Hutter, and Massimiliano Pontil. 2025. "Unlocking State-Tracking in Linear RNNs Through Negative Eigenvalues." *International Conference on Learning Representations*. https://arxiv.org/abs/2411.12537.

Greensmith, Evan, Peter L Bartlett, and Jonathan Baxter. 2004. "Variance Reduction Techniques for Gradient Estimates in Reinforcement Learning." *Journal of Machine Learning Research* 5: 1471–530.

Gretton, Arthur, Karsten M. Borgwardt, Malte J. Rasch, Bernhard Schölkopf, and Alexander Smola. 2012. "A Kernel Two-Sample Test." *Journal of Machine Learning Research* 13: 723–

73.

Griewank, Andreas, and Andrea Walther. 2008. *Evaluating Derivatives: Principles and Techniques of Algorithmic Differentiation*. 2nd ed. Society for Industrial; Applied Mathematics (SIAM). https://doi.org/10.1137/1.9780898717761.

Grishina, Ekaterina, Matvey Smirnov, and Maxim Rakhuba. 2025. "Accelerating Newton–Schulz Iteration for Orthogonalization via Chebyshev-Type Polynomials." *arXiv Preprint arXiv:2506.10935*.

Gu, Albert. 2023. "Modeling Sequences with Structured State Spaces." PhD thesis, Stanford University.

Gu, Albert, and Tri Dao. 2023. "Mamba: Linear-Time Sequence Modeling with Selective State Spaces." *arXiv Preprint arXiv:2312.00752*.

Gu, Albert, Tri Dao, Stefano Ermon, Atri Rudra, and Christopher Ré. 2020. "HiPPO: Recurrent Memory with Optimal Polynomial Projections." *Advances in Neural Information Processing Systems* 33: 1474–87.

Gu, Albert, Karan Goel, and Christopher Ré. 2022. "Efficiently Modeling Long Sequences with Structured State Spaces." *International Conference on Learning Representations*. https://arxiv.org/abs/2111.00396.

Gu, Albert, Ankit Gupta, Karan Goel, and Christopher Ré. 2022. "On the Parameterization and Initialization of Diagonal State Space Models." *Advances in Neural Information Processing Systems* 35: 35971–83.

Gu, Albert, Isys Johnson, Aman Timalsina, Atri Rudra, and Christopher Ré. 2023. "How to Train Your HiPPO: State Space Models with Generalized Orthogonal Basis Projections." *International Conference on Learning Representations*. https://arxiv.org/abs/2206.12037.

Gu, Xiangming, Tianyu Pang, Chao Du, et al. 2025. "When Attention Sink Emerges in Language Models: An Empirical View." *International Conference on Learning Representations*. https://arxiv.org/abs/2410.10781.

Gulrajani, Ishaan, Faruk Ahmed, Martin Arjovsky, Vincent Dumoulin, and Aaron Courville. 2017. "Improved Training of Wasserstein GANs." *Advances in Neural Information Processing Systems* 30.

Guo, Han, Songlin Yang, Tarushii Goel, Eric P. Xing, Tri Dao, and Yoon Kim. 2025. "Log-Linear Attention." *arXiv Preprint arXiv:2506.04761*.

Gupta, Vineet, Tomer Koren, and Yoram Singer. 2018. "Shampoo: Preconditioned Stochastic Tensor Optimization." *Proceedings of the 35th International Conference on Machine Learning (ICML)*, 1842–50.

Gutmann, Michael U., and Aapo Hyvärinen. 2012. "Noise-Contrastive Estimation of Unnormalized Statistical Models, with Applications to Natural Image Statistics." *Journal of Machine Learning Research* 13: 307–61. https://www.jmlr.org/papers/v13/gutmann12a.h

tml.

Haarnoja, Tuomas, Aurick Zhou, Pieter Abbeel, and Sergey Levine. 2018. "Soft Actor-Critic: Off-Policy Maximum Entropy Deep Reinforcement Learning with a Stochastic Actor." *International Conference on Machine Learning*, 1861–70.

Haarnoja, Tuomas, Aurick Zhou, Kristian Hartikainen, et al. 2018. "Soft Actor-Critic Algorithms and Applications." *arXiv Preprint arXiv:1812.05905*.

Hafner, Danijar, Jurgis Pasukonis, Jimmy Ba, and Timothy Lillicrap. 2025. "Mastering Diverse Control Tasks Through World Models." *Nature* 640 (8059): 647–53. https://doi.org/10.1038/s41586-025-08744-2.

Hägele, Alexander, Elie Bakouch, Atli Kosson, Loubna Ben Allal, Leandro von Werra, and Martin Jaggi. 2024. "Scaling Laws and Compute-Optimal Training Beyond Fixed Training Durations." *Advances in Neural Information Processing Systems* 37. https://arxiv.org/abs/2405.18392.

Hang, Tiankai, Shuyang Gu, Chen Li, et al. 2023. "Efficient Diffusion Training via Min-SNR Weighting Strategy." *Proceedings of the IEEE/CVF International Conference on Computer Vision*, 7441–51. https://arxiv.org/abs/2303.09556.

Hansen, Nicklas, Hao Su, and Xiaolong Wang. 2024. "TD-MPC2: Scalable, Robust World Models for Continuous Control." *International Conference on Learning Representations*. https://arxiv.org/abs/2310.16828.

Hansen-Estruch, Philippe, Jiahui Chen, Vivek Ramanujan, et al. 2026. "ViTok-V2: Scaling Native Resolution Auto-Encoders to 5 Billion Parameters." *arXiv Preprint arXiv:2605.05331*.

Hansen-Estruch, Philippe, David Yan, Ching-Yao Chung, et al. 2025. "Learnings from Scaling Visual Tokenizers for Reconstruction and Generation." *arXiv Preprint arXiv:2501.09755*.

Hassani, Ali, Steven Walton, Nikhil Shah, Abulikemu Abuduweili, Jiachen Li, and Humphrey Shi. 2021. "Escaping the Big Data Paradigm with Compact Transformers." *arXiv Preprint arXiv:2104.05704*.

Hasselt, Hado van. 2010. "Double Q-Learning." *Advances in Neural Information Processing Systems* 23: 2613–21.

Hasselt, Hado van, Yotam Doron, Florian Strub, Matteo Hessel, Nicolas Sonnerat, and Joseph Modayil. 2018. "Deep Reinforcement Learning and the Deadly Triad." *arXiv Preprint arXiv:1812.02648*.

Haviv, Adi, Ori Ram, Ofir Press, Peter Izsak, and Omer Levy. 2022. "Transformer Language Models Without Positional Encodings Still Learn Positional Information." *Findings of the Association for Computational Linguistics: EMNLP 2022*. https://arxiv.org/abs/2203.16634.

He, Horace. 2022. *Making Deep Learning Go Brrrr from First Principles*.

Https://horace.io/brrr_intro.html.

He, Kaiming, Xinlei Chen, Saining Xie, Yanghao Li, Piotr Dollár, and Ross Girshick. 2022. "Masked Autoencoders Are Scalable Vision Learners." *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 16000–16009. https://doi.org/10.1109/cvpr52688.2022.01553.

He, Kaiming, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2015. "Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification." *Proceedings of the IEEE International Conference on Computer Vision*, 1026–34. https://doi.org/10.1109/iccv.2015.123.

Heimersheim, Stefan, and Neel Nanda. 2024. "How to Use and Interpret Activation Patching." *arXiv Preprint arXiv:2404.15255*.

Henderson, Peter, Riashat Islam, Philip Bachman, Joelle Pineau, Doina Precup, and David Meger. 2018. "Deep Reinforcement Learning That Matters." *Proceedings of the AAAI Conference on Artificial Intelligence* 32: 3207–14.

Hendrycks, Dan, and Kevin Gimpel. 2016. "Gaussian Error Linear Units (GELUs)." *ArXiv:1606.08415*. https://arxiv.org/abs/1606.08415.

Hennessy, John L, and David A Patterson. 2011. *Computer Architecture: A Quantitative Approach*. Elsevier. https://www.elsevier.com/books/computer-architecture/hennessy/978-0-12-383872-8.

Henry, Alex, Prudhvi Raj Dachapally, Shubham Shantaram Pawar, and Yuxuan Chen. 2020. "Query-Key Normalization for Transformers." *Findings of the Association for Computational Linguistics: EMNLP 2020*.

Heo, Byeongho, Song Park, Dongyoon Han, and Sangdoo Yun. 2024. "Rotary Position Embedding for Vision Transformer." *European Conference on Computer Vision*, 289–305. https://arxiv.org/abs/2403.13298.

Hessel, Jack, Ari Holtzman, Maxwell Forbes, Ronan Le Bras, and Yejin Choi. 2021. "CLIPScore: A Reference-Free Evaluation Metric for Image Captioning." *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*. https://arxiv.org/abs/2104.08718.

Hessel, Matteo, Joseph Modayil, Hado van Hasselt, et al. 2018. "Rainbow: Combining Improvements in Deep Reinforcement Learning." *Proceedings of the AAAI Conference on Artificial Intelligence* 32: 3215–22.

Heusel, Martin, Hubert Ramsauer, Thomas Unterthiner, Bernhard Nessler, and Sepp Hochreiter. 2017. "GANs Trained by a Two Time-Scale Update Rule Converge to a Local Nash Equilibrium." *Advances in Neural Information Processing Systems* 30. https://arxiv.org/abs/1706.08500.

Hillis, W. Daniel, and Guy L. Steele Jr. 1986. "Data Parallel Algorithms." *Communications of the ACM* 29 (12): 1170–83.

Hinton, Geoffrey E. 2002. "Training Products of Experts by Minimizing Contrastive Divergence." *Neural Computation* 14 (8): 1771–800.

Hinton, Geoffrey, Oriol Vinyals, and Jeff Dean. 2015. "Distilling the Knowledge in a Neural Network." *arXiv Preprint arXiv:1503.02531*.

Hiraoka, Takuya, Takahisa Imagawa, Taisei Hashimoto, Takashi Onishi, and Yoshimasa Tsuruoka. 2022. "Dropout Q-Functions for Doubly Efficient Reinforcement Learning." *International Conference on Learning Representations*.

Ho, Jonathan, Ajay Jain, and Pieter Abbeel. 2020. "Denoising Diffusion Probabilistic Models." *Advances in Neural Information Processing Systems* 33: 6840–51. https://arxiv.org/abs/2006.11239.

Ho, Jonathan, and Tim Salimans. 2022. "Classifier-Free Diffusion Guidance." *arXiv Preprint arXiv:2207.12598*.

Ho, Jonathan, Tim Salimans, Alexey Gritsenko, William Chan, Mohammad Norouzi, and David J. Fleet. 2022. "Video Diffusion Models." *Advances in Neural Information Processing Systems* 35. https://arxiv.org/abs/2204.03458.

Hochreiter, Sepp, and Jürgen Schmidhuber. 1997. "Long Short-Term Memory." *Neural Computation* 9 (8): 1735–80. https://doi.org/10.1162/neco.1997.9.8.1735.

Hockney, Roger W. 1994. "The Communication Challenge for MPP: Intel Paragon and Meiko CS-2." *Parallel Computing* 20 (3): 389–98. https://doi.org/10.1016/S0167-8191(06)80021-9.

Hoeffding, Wassily. 1963. "Probability Inequalities for Sums of Bounded Random Variables." *Journal of the American Statistical Association* 58 (301): 13–30.

Hoffmann, Jordan, Sebastian Borgeaud, Arthur Mensch, et al. 2022. "Training Compute-Optimal Large Language Models." *Advances in Neural Information Processing Systems* 35. https://arxiv.org/abs/2203.15556.

Holderrieth, Peter, and Ezra Erives. 2025. "An Introduction to Flow Matching and Diffusion Models." *arXiv Preprint arXiv:2506.02070*. https://arxiv.org/abs/2506.02070.

Hoogeboom, Emiel, Jonathan Heek, and Tim Salimans. 2023. "Simple Diffusion: End-to-End Diffusion for High Resolution Images." *International Conference on Machine Learning*. https://arxiv.org/abs/2301.11093.

Hoogeboom, Emiel, Didrik Nielsen, Priyank Jaini, Patrick Forré, and Max Welling. 2021. "Argmax Flows and Multinomial Diffusion: Learning Categorical Distributions." *Advances in Neural Information Processing Systems* 34. https://arxiv.org/abs/2102.05379.

Hooper, Coleman, Sehoon Kim, Hiva Mohammadzadeh, et al. 2024. "KVQuant: Towards 10 Million Context Length LLM Inference with KV Cache Quantization." *Advances in Neural Information Processing Systems* 37. https://arxiv.org/abs/2401.18079.

Horowitz, Mark. 2014. "1.1 Computing's Energy Problem (and What We Can Do about It)."

IEEE International Solid-State Circuits Conference (ISSCC), 10–14. https://doi.org/10.1109/ISSCC.2014.6757323.

Howard, Ronald A. 1960. *Dynamic Programming and Markov Processes*. The Technology Press of the Massachusetts Institute of Technology; John Wiley & Sons.

Hu, Shengding, Yuge Tu, Xu Han, et al. 2024. "MiniCPM: Unveiling the Potential of Small Language Models with Scalable Training Strategies." *ArXiv:2404.06395*. https://arxiv.org/abs/2404.06395.

Huang, Cheng-Zhi Anna, Ashish Vaswani, Jakob Uszkoreit, et al. 2018. "Music Transformer: Generating Music with Long-Term Structure." *International Conference on Learning Representations*. https://arxiv.org/abs/1809.04281.

Huang, Shengyi, Rousslan F J Dossa, Chang Ye, et al. 2022. "CleanRL: High-Quality Single-File Implementations of Deep Reinforcement Learning Algorithms." *Journal of Machine Learning Research* 23 (274): 1–18.

Huang, Shengyi, Rousslan Fernand Julien Dossa, Antonin Raffin, Anssi Kanervisto, and Weixun Wang. 2022. *The 37 Implementation Details of Proximal Policy Optimization*. ICLR Blog Track, 25 March 2022, https://iclr-blog-track.github.io/2022/03/25/ppo-implementation-details/.

Huang, Xun, and Serge Belongie. 2017. "Arbitrary Style Transfer in Real-Time with Adaptive Instance Normalization." *Proceedings of the IEEE International Conference on Computer Vision*, 1501–10. https://arxiv.org/abs/1703.06868.

Huang, Yanping, Youlong Cheng, Ankur Bapna, et al. 2019. "GPipe: Efficient Training of Giant Neural Networks Using Pipeline Parallelism." *Advances in Neural Information Processing Systems* 32. https://arxiv.org/abs/1811.06965.

Huang, Yiwen, Aaron Gokaslan, Volodymyr Kuleshov, and James Tompkin. 2024. "The GAN Is Dead; Long Live the GAN! A Modern GAN Baseline." *Advances in Neural Information Processing Systems* 37.

Hyvärinen, Aapo. 2005. "Estimation of Non-Normalized Statistical Models by Score Matching." *Journal of Machine Learning Research* 6: 695–709.

Ilic, Aleksandar, Frederico Pratas, and Leonel Sousa. 2014. "Cache-Aware Roofline Model: Upgrading the Loft." *IEEE Computer Architecture Letters* 13 (1): 21–24. https://doi.org/10.1109/L-CA.2013.6.

Isola, Phillip, Jun-Yan Zhu, Tinghui Zhou, and Alexei A Efros. 2017. "Image-to-Image Translation with Conditional Adversarial Networks." *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 1125–34. https://doi.org/10.1109/cvpr.2017.632.

Ivanov, Andrei, Nikoli Dryden, Tal Ben-Nun, Shigang Li, and Torsten Hoefler. 2021. "Data Movement Is All You Need: A Case Study on Optimizing Transformers." *Proceedings of Machine Learning and Systems* 3. https://arxiv.org/abs/2007.00072.

Izmailov, Pavel, Dmitrii Podoprikhin, Timur Garipov, Dmitry Vetrov, and Andrew Gordon Wilson. 2018. "Averaging Weights Leads to Wider Optima and Better Generalization." *Uncertainty in Artificial Intelligence*, 876–85. https://arxiv.org/abs/1803.05407.

Jaakkola, Tommi, Michael I Jordan, and Satinder P Singh. 1994. "On the Convergence of Stochastic Iterative Dynamic Programming Algorithms." *Neural Computation* 6 (6): 1185–201.

Jacobs, Robert A., Michael I. Jordan, Steven J. Nowlan, and Geoffrey E. Hinton. 1991. "Adaptive Mixtures of Local Experts." *Neural Computation* 3 (1): 79–87.

Jacot, Arthur, Franck Gabriel, and Clément Hongler. 2018. "Neural Tangent Kernel: Convergence and Generalization in Neural Networks." *Advances in Neural Information Processing Systems* 31. https://doi.org/10.1145/3406325.3465355.

Jaegle, Andrew, Sebastian Borgeaud, Jean-Baptiste Alayrac, et al. 2022. "Perceiver IO: A General Architecture for Structured Inputs & Outputs." *International Conference on Learning Representations*.

Jaegle, Andrew, Felix Gimeno, Andy Brock, Oriol Vinyals, Andrew Zisserman, and Joao Carreira. 2021. "Perceiver: General Perception with Iterative Attention." *International Conference on Machine Learning*, 4651–64.

Jain, Paras, Ajay Jain, Aniruddha Nrusimha, et al. 2020. "Checkmate: Breaking the Memory Wall with Optimal Tensor Rematerialization." *Proceedings of Machine Learning and Systems* 2. https://arxiv.org/abs/1910.02653.

Jain, Sarthak, and Byron C. Wallace. 2019. "Attention Is Not Explanation." *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics*, 3543–56.

Jamba Team. 2024. "Jamba-1.5: Hybrid Transformer-Mamba Models at Scale." *arXiv Preprint arXiv:2408.12570*.

Janner, Michael, Justin Fu, Marvin Zhang, and Sergey Levine. 2019. "When to Trust Your Model: Model-Based Policy Optimization." *Advances in Neural Information Processing Systems* 32.

Jayasumana, Sadeep, Srikumar Ramalingam, Andreas Veit, Daniel Glasner, Ayan Chakrabarti, and Sanjiv Kumar. 2024. "Rethinking FID: Towards a Better Evaluation Metric for Image Generation." *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 9307–15. https://arxiv.org/abs/2401.09603.

Jelassi, Samy, David Brandfonbrener, Sham M. Kakade, and Eran Malach. 2024. "Repeat After Me: Transformers Are Better Than State Space Models at Copying." *International Conference on Machine Learning*, 21502–21. https://arxiv.org/abs/2402.01032.

Jiang, Albert Q., Alexandre Sablayrolles, Arthur Mensch, et al. 2023. "Mistral 7B." *arXiv Preprint arXiv:2310.06825*.

Jiang, Albert Q., Alexandre Sablayrolles, Antoine Roux, et al. 2024. "Mixtral of Experts." *arXiv Preprint arXiv:2401.04088*.

Jiang, Nan, Alex Kulesza, Satinder Singh, and Richard Lewis. 2015. "The Dependence of Effective Planning Horizon on Model Accuracy." *International Conference on Autonomous Agents and Multiagent Systems*.

Jiang, Nick, Amil Dravid, Alexei Efros, and Yossi Gandelsman. 2025. "Vision Transformers Don't Need Trained Registers." *Advances in Neural Information Processing Systems*. https://arxiv.org/abs/2506.08010.

Jin, Chi, Zeyuan Allen-Zhu, Sébastien Bubeck, and Michael I. Jordan. 2018. "Is Q-Learning Provably Efficient?" *Advances in Neural Information Processing Systems* 31: 4863–73. https://arxiv.org/abs/1807.03765.

Jin, Ying, Zhuoran Yang, and Zhaoran Wang. 2021. "Is Pessimism Provably Efficient for Offline RL?" *International Conference on Machine Learning*, Proceedings of machine learning research, vol. 139: 5084–96.

Jolicoeur-Martineau, Alexia. 2019. "The Relativistic Discriminator: A Key Element Missing from Standard GAN." *International Conference on Learning Representations*.

Jolicoeur-Martineau, Alexia. 2020. "On Relativistic f-Divergences." *Proceedings of the 37th International Conference on Machine Learning*, 4931–39.

Jordan, Keller. 2026. *Modded-NanoGPT Optimization Benchmark*. Https://github.com/KellerJordan/modded-nanogpt/tree/master/records/track_3_optimization.

Jordan, Keller, and contributors. 2024. *Modded-Nanogpt: Speedrunning the NanoGPT Baseline*. Https://github.com/KellerJordan/modded-nanogpt.

Jordan, Keller, Yuchen Jin, Vlado Boza, et al. 2024. *Muon: An Optimizer for Hidden Layers in Neural Networks*. Https://kellerjordan.github.io/posts/muon/.

Jouppi, Norman P, Cliff Young, Nishant Patil, et al. 2017. "In-Datacenter Performance Analysis of a Tensor Processing Unit." *2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA)*, 1–12. https://doi.org/10.1145/3140659.3080246.

Jozefowicz, Rafal, Wojciech Zaremba, and Ilya Sutskever. 2015. "An Empirical Exploration of Recurrent Network Architectures." *International Conference on Machine Learning*, 2342–50. https://proceedings.mlr.press/v37/jozefowicz15.html.

Kaddour, Jean. 2022. "Stop Wasting My Time! Saving Days of ImageNet and BERT Training with Latest Weight Averaging." *arXiv Preprint arXiv:2209.14981*.

Kadkhodaie, Zahra, Florentin Guth, Eero P. Simoncelli, and Stéphane Mallat. 2024. "Generalization in Diffusion Models Arises from Geometry-Adaptive Harmonic Representations." *International Conference on Learning Representations*. https://arxiv.org/abs/2310.02557.

Kaelbling, Leslie Pack, Michael L Littman, and Anthony R Cassandra. 1998. "Planning and Acting in Partially Observable Stochastic Domains." *Artificial Intelligence* 101 (1–2): 99–134. https://doi.org/10.1016/S0004-3702(98)00023-X.

Kahng, Minsuk, Nikhil Thorat, Duen Horng Chau, Fernanda B. Viégas, and Martin Wattenberg. 2018. "GAN Lab: Understanding Complex Deep Generative Models Using Interactive Visual Experimentation." *IEEE Transactions on Visualization and Computer Graphics* 25 (1): 310–20.

Kakade, Sham, and John Langford. 2002. "Approximately Optimal Approximate Reinforcement Learning." *International Conference on Machine Learning*, 267–74.

Kalchbrenner, Nal, Edward Grefenstette, and Phil Blunsom. 2014. "A Convolutional Neural Network for Modelling Sentences." *ArXiv:1404.2188*. https://arxiv.org/abs/1404.2188.

Kalra, Dayal Singh, and Maissam Barkeshli. 2024. "Why Warmup the Learning Rate? Underlying Mechanisms and Improvements." *Advances in Neural Information Processing Systems* 37. https://arxiv.org/abs/2406.09405.

Kamb, Mason, and Surya Ganguli. 2025. "An Analytic Theory of Creativity in Convolutional Diffusion Models." *Proceedings of the 42nd International Conference on Machine Learning*, Proceedings of machine learning research, vol. 267: 28795–831.

Kang, Minguk, Jun-Yan Zhu, Richard Zhang, et al. 2023. "Scaling up GANs for Text-to-Image Synthesis." *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 10124–34. https://arxiv.org/abs/2303.05511.

Kaplan, Jared, Sam McCandlish, Tom Henighan, et al. 2020. "Scaling Laws for Neural Language Models." *ArXiv:2001.08361*. https://arxiv.org/abs/2001.08361.

Karras, Tero, Timo Aila, Samuli Laine, and Jaakko Lehtinen. 2018. "Progressive Growing of GANs for Improved Quality, Stability, and Variation." *International Conference on Learning Representations*. https://arxiv.org/abs/1710.10196.

Karras, Tero, Miika Aittala, Timo Aila, and Samuli Laine. 2022. "Elucidating the Design Space of Diffusion-Based Generative Models." *Advances in Neural Information Processing Systems* 35.

Karras, Tero, Miika Aittala, Janne Hellsten, Samuli Laine, Jaakko Lehtinen, and Timo Aila. 2020. "Training Generative Adversarial Networks with Limited Data." *Advances in Neural Information Processing Systems* 33.

Karras, Tero, Miika Aittala, Tuomas Kynkäänniemi, Jaakko Lehtinen, Timo Aila, and Samuli Laine. 2024. "Guiding a Diffusion Model with a Bad Version of Itself." *Advances in Neural Information Processing Systems* 37. https://arxiv.org/abs/2406.02507.

Karras, Tero, Miika Aittala, Samuli Laine, et al. 2021. "Alias-Free Generative Adversarial Networks." *Advances in Neural Information Processing Systems* 34: 852–63. https://arxiv.org/abs/2106.12423.

Karras, Tero, Miika Aittala, Jaakko Lehtinen, Janne Hellsten, Timo Aila, and Samuli Laine. 2024. "Analyzing and Improving the Training Dynamics of Diffusion Models." *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*.

Karras, Tero, Samuli Laine, and Timo Aila. 2019. "A Style-Based Generator Architecture for Generative Adversarial Networks." *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 4401–10. https://arxiv.org/abs/1812.04948.

Karras, Tero, Samuli Laine, Miika Aittala, Janne Hellsten, Jaakko Lehtinen, and Timo Aila. 2020. "Analyzing and Improving the Image Quality of StyleGAN." *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 8110–19. https://arxiv.org/abs/1912.04958.

Kasimbeg, Priya, Frank Schneider, Runa Eschenhagen, et al. 2025. "Accelerating Neural Network Training: An Analysis of the AlgoPerf Competition." *International Conference on Learning Representations*. https://arxiv.org/abs/2502.15015.

Katharopoulos, Angelos, Apoorv Vyas, Nikolaos Pappas, and François Fleuret. 2020. "Transformers Are RNNs: Fast Autoregressive Transformers with Linear Attention." *International Conference on Machine Learning*.

Kawar, Bahjat, Michael Elad, Stefano Ermon, and Jiaming Song. 2022. "Denoising Diffusion Restoration Models." *Advances in Neural Information Processing Systems* 35. https://arxiv.org/abs/2201.11793.

Kazemnejad, Amirhossein, Milad Aghajohari, Eva Portelance, et al. 2025. "VinePPO: Refining Credit Assignment in RL Training of LLMs." *International Conference on Machine Learning*. https://arxiv.org/abs/2410.01679.

Kazemnejad, Amirhossein, Inkit Padhi, Karthikeyan Natesan Ramamurthy, Payel Das, and Siva Reddy. 2023. "The Impact of Positional Encoding on Length Generalization in Transformers." *Advances in Neural Information Processing Systems* 36. https://arxiv.org/abs/2305.19466.

Keskar, Nitish Shirish, Dheevatsa Mudigere, Jorge Nocedal, Mikhail Smelyanskiy, and Ping Tak Peter Tang. 2017. "On Large-Batch Training for Deep Learning: Generalization Gap and Sharp Minima." *International Conference on Learning Representations*. https://arxiv.org/abs/1609.04836.

Khaled, Ahmed, Kaan Ozkara, Tao Yu, Mingyi Hong, and Youngsuk Park. 2025. "MuonBP: Faster Muon via Block-Periodic Orthogonalization." *arXiv Preprint arXiv:2510.16981*.

Kidambi, Rahul, Aravind Rajeswaran, Praneeth Netrapalli, and Thorsten Joachims. 2020. "MOReL: Model-Based Offline Reinforcement Learning." *Advances in Neural Information Processing Systems* 33.

Kim, Dongjun, Chieh-Hsin Lai, Wei-Hsiang Liao, et al. 2024. "Consistency Trajectory Models: Learning Probability Flow ODE Trajectory of Diffusion." *International Conference on Learning Representations*. https://arxiv.org/abs/2310.02279.

Kim, Gyu Yeol, and Min-hwan Oh. 2026. "Convergence of Muon with Newton–Schulz." *International Conference on Learning Representations*.

Kim, Jaeyeon, Kulin Shah, Vasilis Kontonis, Sham Kakade, and Sitan Chen. 2025. "Train for the Worst, Plan for the Best: Understanding Token Ordering in Masked Diffusions." *International Conference on Machine Learning*.

Kimi Team. 2025a. "Kimi K2: Open Agentic Intelligence." *arXiv Preprint arXiv:2507.20534*.

Kimi Team. 2025b. "Kimi Linear: An Expressive, Efficient Attention Architecture." *arXiv Preprint arXiv:2510.26692*.

Kingma, Diederik P, and Jimmy Ba. 2015. "Adam: A Method for Stochastic Optimization." *International Conference on Learning Representations*. https://arxiv.org/abs/1412.6980.

Kingma, Diederik P., and Ruiqi Gao. 2023. "Understanding Diffusion Objectives as the ELBO with Simple Data Augmentation." *Advances in Neural Information Processing Systems* 36. https://arxiv.org/abs/2303.00848.

Kingma, Diederik P., Tim Salimans, Ben Poole, and Jonathan Ho. 2021. "Variational Diffusion Models." *Advances in Neural Information Processing Systems* 34.

Kingma, Diederik P., and Max Welling. 2014. "Auto-Encoding Variational Bayes." *International Conference on Learning Representations (ICLR)*. https://arxiv.org/abs/1312.6114.

Kitaev, Nikita, Lukasz Kaiser, and Anselm Levskaya. 2020. "Reformer: The Efficient Transformer." *International Conference on Learning Representations*.

Koehler, Frederic, Alexander Heckett, and Andrej Risteski. 2022. "Statistical Efficiency of Score Matching: The View from Isoperimetry." *arXiv Preprint arXiv:2210.00726*. https://arxiv.org/abs/2210.00726.

Konda, Vijay R, and John N Tsitsiklis. 2000. "Actor-Critic Algorithms." *Advances in Neural Information Processing Systems* 12.

Kong, Jungil, Jaehyeon Kim, and Jaekyoung Bae. 2020. "HiFi-GAN: Generative Adversarial Networks for Efficient and High Fidelity Speech Synthesis." *Advances in Neural Information Processing Systems* 33.

Kool, Wouter, Herke van Hoof, and Max Welling. 2019. "Buy 4 REINFORCE Samples, Get a Baseline for Free!" *ICLR 2019 Workshop on Deep Reinforcement Learning Meets Structured Prediction*. https://mlanthology.org/iclrw/2019/kool2019iclrw-buy/.

Korbak, Tomasz, Ethan Perez, and Christopher L Buckley. 2022. "RL with KL Penalties Is Better Viewed as Bayesian Inference." *Findings of the Association for Computational Linguistics: EMNLP 2022*. https://arxiv.org/abs/2205.11275.

Korthikanti, Vijay, Jared Casper, Sangkug Lym, et al. 2022. "Reducing Activation Recomputation in Large Transformer Models." *arXiv Preprint arXiv:2205.05198*.

Kosson, Atli, Bettina Messmer, and Martin Jaggi. 2024. "Rotational Equilibrium: How Weight Decay Balances Learning Across Neural Networks." *International Conference on Machine Learning*. https://arxiv.org/abs/2305.17212.

Kosson, Atli, Jeremy Welborn, Yang Liu, Martin Jaggi, and Xi Chen. 2025. "Weight Decay May Matter More Than muP for Learning Rate Transfer in Practice." *arXiv Preprint arXiv:2510.19093*.

Kostrikov, Ilya, Ashvin Nair, and Sergey Levine. 2022. "Offline Reinforcement Learning with Implicit Q-Learning." *International Conference on Learning Representations*. https://arxiv.org/abs/2110.06169.

Kovalev, Dmitry. 2025. "Understanding Gradient Orthogonalization for Deep Learning via Non-Euclidean Trust-Region Optimization." *arXiv Preprint arXiv:2503.12645*.

Krajewski, Jakub, Jan Ludziejewski, Kamil Adamczewski, et al. 2024. "Scaling Laws for Fine-Grained Mixture of Experts." *International Conference on Machine Learning*, 33270–88. https://arxiv.org/abs/2402.07871.

Krakovna, Victoria, Jonathan Uesato, Vladimir Mikulik, et al. 2020. *Specification Gaming: The Flip Side of AI Ingenuity*. DeepMind blog, 21 April 2020, https://deepmind.google/blog/specification-gaming-the-flip-side-of-ai-ingenuity/.

Kramár, János, Tom Lieberum, Rohin Shah, and Neel Nanda. 2024. "AtP*: An Efficient and Scalable Method for Localizing LLM Behaviour to Components." *arXiv Preprint arXiv:2403.00745*.

Kramers, Hendrik A. 1940. "Brownian Motion in a Field of Force and the Diffusion Model of Chemical Reactions." *Physica* 7 (4): 284–304.

Krizhevsky, Alex, Ilya Sutskever, and Geoffrey E Hinton. 2012. "ImageNet Classification with Deep Convolutional Neural Networks." *Advances in Neural Information Processing Systems*, 1097–105. https://doi.org/10.5555/2999134.2999257.

Kuleshov, Volodymyr. 2023. *Deep Generative Models: Lecture Slides (Lectures 10–13: Advanced Topics in GANs, Energy-Based Models, Score-Based Generative Models, Diffusion Models)*. Course materials, Cornell Tech. Course website: https://kuleshov-group.github.io/dgm-website/; slides: https://github.com/kuleshov-group/dgm-resources.

Kullback, Solomon. 1959. *Information Theory and Statistics*. John Wiley & Sons.

Kumar, Aviral, Aurick Zhou, George Tucker, and Sergey Levine. 2020. "Conservative Q-Learning for Offline Reinforcement Learning." *Advances in Neural Information Processing Systems* 33.

Kumar, Tanishq, Zachary Ankner, Benjamin F. Spector, et al. 2025. "Scaling Laws for Precision." *International Conference on Learning Representations*. https://arxiv.org/abs/2411.04330.

Kung, Sun Yuan. 1988. "VLSI Array Processors." *Prentice Hall*.

Kunstner, Frederik, Jacques Chen, Jonathan Wilder Lavington, and Mark Schmidt. 2023. "Noise Is Not the Main Factor Behind the Gap Between SGD and Adam on Transformers, but Sign Descent Might Be." *International Conference on Learning Representations*. https://arxiv.org/abs/2304.13960.

Kunstner, Frederik, Robin Yadav, Alan Milligan, Mark Schmidt, and Alberto Bietti. 2024. "Heavy-Tailed Class Imbalance and Why Adam Outperforms Gradient Descent on Language Models." *Advances in Neural Information Processing Systems* 37. https://arxiv.org/abs/2402.19449.

Kwon, Woosuk, Zhuohan Li, Siyuan Zhuang, et al. 2023. "Efficient Memory Management for Large Language Model Serving with PagedAttention." *Proceedings of the 29th Symposium on Operating Systems Principles*, 611–26. https://doi.org/10.1145/3600006.3613165.

Kynkäänniemi, Tuomas, Miika Aittala, Tero Karras, Samuli Laine, Timo Aila, and Jaakko Lehtinen. 2024. "Applying Guidance in a Limited Interval Improves Sample and Distribution Quality in Diffusion Models." *Advances in Neural Information Processing Systems* 37. https://arxiv.org/abs/2404.07724.

Kynkäänniemi, Tuomas, Tero Karras, Miika Aittala, Timo Aila, and Jaakko Lehtinen. 2023. "The Role of ImageNet Classes in Fréchet Inception Distance." *International Conference on Learning Representations*. https://arxiv.org/abs/2203.06026.

Kynkäänniemi, Tuomas, Tero Karras, Samuli Laine, Jaakko Lehtinen, and Timo Aila. 2019. "Improved Precision and Recall Metric for Assessing Generative Models." *Advances in Neural Information Processing Systems* 32. https://arxiv.org/abs/1904.06991.

Lahoti, Aakash, Kevin Y. Li, Berlin Chen, et al. 2026. "Mamba-3: Improved Sequence Modeling Using State Space Principles." *arXiv Preprint arXiv:2603.15569*.

Lai, Chieh-Hsin, Yang Song, Dongjun Kim, Yuki Mitsufuji, and Stefano Ermon. 2025. "The Principles of Diffusion Models." *arXiv Preprint arXiv:2510.21890*. https://arxiv.org/abs/2510.21890.

Lai, Tze Leung, and Herbert Robbins. 1985. "Asymptotically Efficient Adaptive Allocation Rules." *Advances in Applied Mathematics* 6 (1): 4–22. https://doi.org/10.1016/0196-8858(85)90002-8.

Lambert, Nathan. 2026. *Reinforcement Learning from Human Feedback*. Online book, continuously updated since 2024; arXiv:2504.12501; print edition Manning/Simon & Schuster, https://rlhfbook.com/.

Lattimore, Tor, and Csaba Szepesvári. 2020. *Bandit Algorithms*. Cambridge University Press. https://tor-lattimore.com/downloads/book/book.pdf.

Lee, Jason D., Max Simchowitz, Michael I. Jordan, and Benjamin Recht. 2016. "Gradient Descent Only Converges to Minimizers." *29th Annual Conference on Learning Theory*, Proceedings of machine learning research, vol. 49: 1246–57.

Lee, Sang-gil, Wei Ping, Boris Ginsburg, Bryan Catanzaro, and Sungroh Yoon. 2023. "BigV-

GAN: A Universal Neural Vocoder with Large-Scale Training." *International Conference on Learning Representations*.

Lee, Yongjoon, and Jung-Woo Choi. 2026. "RAF: Relativistic Adversarial Feedback for Universal Speech Synthesis." *Interspeech*.

Lepikhin, Dmitry, HyoukJoong Lee, Yuanzhong Xu, et al. 2021. "GShard: Scaling Giant Models with Conditional Computation and Automatic Sharding." *International Conference on Learning Representations*.

Leviathan, Yaniv, Matan Kalman, and Yossi Matias. 2023. "Fast Inference from Transformers via Speculative Decoding." *International Conference on Machine Learning*, 19274–86.

Levine, Sergey. 2018. "Reinforcement Learning and Control as Probabilistic Inference: Tutorial and Review." *arXiv Preprint arXiv:1805.00909*.

Levine, Sergey, Aviral Kumar, George Tucker, and Justin Fu. 2020. "Offline Reinforcement Learning: Tutorial, Review, and Perspectives on Open Problems." *arXiv Preprint arXiv:2005.01643*.

Lewis, Mike, Yinhan Liu, Naman Goyal, et al. 2019. "BART: Denoising Sequence-to-Sequence Pre-Training for Natural Language Generation, Translation, and Comprehension." *ArXiv:1910.13461*. https://arxiv.org/abs/1910.13461.

Li, Chun-Liang, Wei-Cheng Chang, Yu Cheng, Yiming Yang, and Barnabás Póczos. 2017. "MMD GAN: Towards Deeper Understanding of Moment Matching Network." *Advances in Neural Information Processing Systems* 30.

Li, Gen, Yuting Wei, Yuejie Chi, Yuantao Gu, and Yuxin Chen. 2021. "Softmax Policy Gradient Methods Can Take Exponential Time to Converge." *Proceedings of the 34th Conference on Learning Theory*, Proceedings of machine learning research, vol. 134: 3107–10.

Li, Hao, Zheng Xu, Gavin Taylor, Christoph Studer, and Tom Goldstein. 2018. "Visualizing the Loss Landscape of Neural Nets." *Advances in Neural Information Processing Systems* 31: 6389–99.

Li, Junnan, Dongxu Li, Silvio Savarese, and Steven Hoi. 2023. "BLIP-2: Bootstrapping Language-Image Pre-Training with Frozen Image Encoders and Large Language Models." *International Conference on Machine Learning*, 19730–42.

Li, Ke, and Jitendra Malik. 2018. "Implicit Maximum Likelihood Estimation." *arXiv Preprint arXiv:1809.09087*.

Li, Mu, David G Andersen, Jun Woo Park, et al. 2014. "Scaling Distributed Machine Learning with the Parameter Server." *11th Symposium on Operating Systems Design and Implementation (OSDI 14)*, 583–98. https://doi.org/10.1145/2640087.2644155.

Li, Shen, Yanli Zhao, Rohan Varma, et al. 2020. "PyTorch Distributed: Experiences on Accelerating Data Parallel Training." *Proceedings of the VLDB Endowment* 13 (12): 3005–18. https://doi.org/10.14778/3415478.3415530.

Li, Tianhong, and Kaiming He. 2026. "Back to Basics: Let Denoising Generative Models Denoise." *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 36115–25. https://arxiv.org/abs/2511.13720.

Li, Yucheng, Huiqiang Jiang, Qianhui Wu, et al. 2025. "SCBench: A KV Cache-Centric Analysis of Long-Context Methods." *International Conference on Learning Representations*. https://arxiv.org/abs/2412.10319.

Li, Yuhong, Yingbing Huang, Bowen Yang, et al. 2024. "SnapKV: LLM Knows What You Are Looking for Before Generation." *Advances in Neural Information Processing Systems* 37. https://arxiv.org/abs/2404.14469.

Li, Yuhui, Fangyun Wei, Chao Zhang, and Hongyang Zhang. 2024. "EAGLE: Speculative Sampling Requires Rethinking Feature Uncertainty." *International Conference on Machine Learning*, 28935–48. https://arxiv.org/abs/2401.15077.

Li, Yujia, Kevin Swersky, and Richard Zemel. 2015. "Generative Moment Matching Networks." *Proceedings of the 32nd International Conference on Machine Learning*, 1718–27.

Li, Zichong, Liming Liu, Chen Liang, Weizhu Chen, and Tuo Zhao. 2026. "NorMuon: Making Muon More Efficient and Scalable." *International Conference on Machine Learning*.

Lieber, Opher, Barak Lenz, Hofit Bata, et al. 2024. "Jamba: A Hybrid Transformer-Mamba Language Model." *arXiv Preprint arXiv:2403.19887*.

Lightman, Hunter, Vineet Kosaraju, Yuri Burda, et al. 2024. "Let's Verify Step by Step." *International Conference on Learning Representations*.

Lillicrap, Timothy P, Jonathan J Hunt, Alexander Pritzel, et al. 2016. "Continuous Control with Deep Reinforcement Learning." *International Conference on Learning Representations*. https://arxiv.org/abs/1509.02971.

Lim, Jae Hyun, and Jong Chul Ye. 2017. "Geometric GAN." *arXiv Preprint arXiv:1705.02894*.

Lin, Long-Ji. 1992. "Self-Improving Reactive Agents Based on Reinforcement Learning, Planning and Teaching." *Machine Learning* 8: 293–321.

Lin, Shanchuan, Bingchen Liu, Jiashi Li, and Xiao Yang. 2024. "Common Diffusion Noise Schedules and Sample Steps Are Flawed." *Proceedings of the IEEE/CVF Winter Conference on Applications of Computer Vision*, 5404–11. https://arxiv.org/abs/2305.08891.

Lin, Shanchuan, Anran Wang, and Xiao Yang. 2024. "SDXL-Lightning: Progressive Adversarial Diffusion Distillation." *arXiv Preprint arXiv:2402.13929*.

Lin, Shanchuan, Xin Xia, Yuxi Ren, Ceyuan Yang, Xuefeng Xiao, and Lu Jiang. 2025. "Diffusion Adversarial Post-Training for One-Step Video Generation." *Proceedings of the 42nd International Conference on Machine Learning*.

Lin, Zhouhan, Minwei Feng, Cicero Nogueira dos Santos, et al. 2017. "A Structured Self-Attentive Sentence Embedding." *ArXiv:1703.03130*. https://arxiv.org/abs/1703.03130.

Lipman, Yaron, Ricky T. Q. Chen, Heli Ben-Hamu, Maximilian Nickel, and Matt Le. 2023. "Flow Matching for Generative Modeling." *International Conference on Learning Representations*.

Lipman, Yaron, Marton Havasi, Peter Holderrieth, et al. 2024. "Flow Matching Guide and Code." *arXiv Preprint arXiv:2412.06264*. https://arxiv.org/abs/2412.06264.

Little, John D. C. 1961. "A Proof for the Queuing Formula: $L = \lambda W$." *Operations Research* 9 (3): 383–87. https://doi.org/10.1287/opre.9.3.383.

Liu, Bo, Rui Wang, Lemeng Wu, Yihao Feng, Peter Stone, and Qiang Liu. 2024. "Longhorn: State Space Models Are Amortized Online Learners." *arXiv Preprint arXiv:2407.14207*.

Liu, Dong C, and Jorge Nocedal. 1989. "On the Limited Memory BFGS Method for Large Scale Optimization." *Mathematical Programming* 45 (1): 503–28. https://doi.org/10.1007/bf01589116.

Liu, Hong, Zhiyuan Li, David Hall, Percy Liang, and Tengyu Ma. 2024. "Sophia: A Scalable Stochastic Second-Order Optimizer for Language Model Pre-Training." *International Conference on Learning Representations*. https://arxiv.org/abs/2305.14342.

Liu, Jie, Gongye Liu, Jiajun Liang, et al. 2025. "Flow-GRPO: Training Flow Matching Models via Online RL." *Advances in Neural Information Processing Systems*. https://arxiv.org/abs/2505.05470.

Liu, Jingyuan, Jianlin Su, Xingcheng Yao, et al. 2025. "Muon Is Scalable for LLM Training." *arXiv Preprint arXiv:2502.16982*.

Liu, Nelson F., Kevin Lin, John Hewitt, et al. 2024. "Lost in the Middle: How Language Models Use Long Contexts." *Transactions of the Association for Computational Linguistics* 12: 157–73. https://arxiv.org/abs/2307.03172.

Liu, Xingchao, Chengyue Gong, and Qiang Liu. 2023. "Flow Straight and Fast: Learning to Generate and Transfer Data with Rectified Flow." *International Conference on Learning Representations*.

Liu, Yue, Yunjie Tian, Yuzhong Zhao, et al. 2024. "VMamba: Visual State Space Model." *Advances in Neural Information Processing Systems*. https://arxiv.org/abs/2401.10166.

Liu, Ze, Yutong Lin, Yue Cao, et al. 2021. "Swin Transformer: Hierarchical Vision Transformer Using Shifted Windows." *Proceedings of the IEEE/CVF International Conference on Computer Vision*, 10012–22. https://doi.org/10.1109/iccv48922.2021.00986.

Liu, Zichen, Changyu Chen, Wenjun Li, et al. 2025. "Understanding R1-Zero-Like Training: A Critical Perspective." *Second Conference on Language Modeling*. https://arxiv.org/abs/2503.20783.

Liu, Zirui, Jiayi Yuan, Hongye Jin, et al. 2024. "KIVI: A Tuning-Free Asymmetric 2bit Quantization for KV Cache." *International Conference on Machine Learning*, 32332–44. https://arxiv.org/abs/2402.02750.

Lopez-Paz, David, and Maxime Oquab. 2017. “Revisiting Classifier Two-Sample Tests.” *International Conference on Learning Representations*.

Loshchilov, Ilya, and Frank Hutter. 2016. “SGDR: Stochastic Gradient Descent with Warm Restarts.” *ArXiv:1608.03983*. https://arxiv.org/abs/1608.03983.

Loshchilov, Ilya, and Frank Hutter. 2019. “Decoupled Weight Decay Regularization.” *International Conference on Learning Representations*. https://arxiv.org/abs/1711.05101.

Lou, Aaron, Chenlin Meng, and Stefano Ermon. 2024. “Discrete Diffusion Modeling by Estimating the Ratios of the Data Distribution.” *Proceedings of the 41st International Conference on Machine Learning*, Proceedings of machine learning research, vol. 235: 32819–48. https://arxiv.org/abs/2310.16834.

Lu, Cheng, and Yang Song. 2025. “Simplifying, Stabilizing and Scaling Continuous-Time Consistency Models.” *International Conference on Learning Representations*.

Lu, Cheng, Kaiwen Zheng, Fan Bao, Jianfei Chen, Chongxuan Li, and Jun Zhu. 2022. “Maximum Likelihood Training for Score-Based Diffusion ODEs by High-Order Denoising Score Matching.” *International Conference on Machine Learning*.

Lu, Cheng, Yuhao Zhou, Fan Bao, Jianfei Chen, Chongxuan Li, and Jun Zhu. 2022. “DPM-Solver: A Fast ODE Solver for Diffusion Probabilistic Model Sampling in Around 10 Steps.” *Advances in Neural Information Processing Systems* 35.

Lu, Cheng, Yuhao Zhou, Fan Bao, Jianfei Chen, Chongxuan Li, and Jun Zhu. 2025. “DPM-Solver++: Fast Solver for Guided Sampling of Diffusion Probabilistic Models.” *Machine Intelligence Research* 22 (4): 730–51. https://doi.org/10.1007/s11633-025-1562-4.

Lu, Enzhe, Zhejun Jiang, Jingyuan Liu, et al. 2025. “MoBA: Mixture of Block Attention for Long-Context LLMs.” *Advances in Neural Information Processing Systems*. https://arxiv.org/abs/2502.13189.

Lucic, Mario, Karol Kurach, Marcin Michalski, Sylvain Gelly, and Olivier Bousquet. 2018. “Are GANs Created Equal? A Large-Scale Study.” *Advances in Neural Information Processing Systems* 31. https://arxiv.org/abs/1711.10337.

Ludziejewski, Jan, Maciej Pióro, Jakub Krajewski, et al. 2025. “Joint MoE Scaling Laws: Mixture of Experts Can Be Memory Efficient.” *arXiv Preprint arXiv:2502.05172*.

Lugmayr, Andreas, Martin Danelljan, Andres Romero, Fisher Yu, Radu Timofte, and Luc Van Gool. 2022. “RePaint: Inpainting Using Denoising Diffusion Probabilistic Models.” *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 11461–71. https://arxiv.org/abs/2201.09865.

Luo, Calvin. 2022. “Understanding Diffusion Models: A Unified Perspective.” *arXiv Preprint arXiv:2208.11970*.

Luo, Tianze, Xingchen Miao, and Wenbo Duan. 2025. “WaveFM: A High-Fidelity and Efficient Vocoder Based on Flow Matching.” *Proceedings of the 2025 Conference of the Nations*

of the Americas Chapter of the Association for Computational Linguistics.

Luo, Weile, Ruibo Fan, Zeyu Li, Dayou Du, Qiang Wang, and Xiaowen Chu. 2024. "Benchmarking and Dissecting the Nvidia Hopper GPU Architecture." *arXiv Preprint arXiv:2402.13499*.

Lyle, Clare, Pablo Samuel Castro, and Marc G. Bellemare. 2019. "A Comparative Analysis of Expected and Distributional Reinforcement Learning." *Proceedings of the AAAI Conference on Artificial Intelligence* 33: 4504–11.

Mack, Yue-Pok, and Bernard W Silverman. 1982. "Weak and Strong Uniform Consistency of Kernel Regression Estimates." *Zeitschrift für Wahrscheinlichkeitstheorie Und Verwandte Gebiete* 61 (3): 405–15. https://doi.org/10.1007/bf00539840.

Malladi, Sadhika, Kaifeng Lyu, Abhishek Panigrahi, and Sanjeev Arora. 2022. "On the SDEs and Scaling Rules for Adaptive Gradient Algorithms." *Advances in Neural Information Processing Systems* 35. https://arxiv.org/abs/2205.10287.

Mania, Horia, Aurelia Guy, and Benjamin Recht. 2018. "Simple Random Search of Static Linear Policies Is Competitive for Reinforcement Learning." *Advances in Neural Information Processing Systems* 31.

Mao, Xudong, Qing Li, Haoran Xie, Raymond Y. K. Lau, Zhen Wang, and Stephen Paul Smolley. 2017. "Least Squares Generative Adversarial Networks." *Proceedings of the IEEE International Conference on Computer Vision*, 2794–802.

Marcus, Mitchell P., Beatrice Santorini, and Mary Ann Marcinkiewicz. 1993. "Building a Large Annotated Corpus of English: The Penn Treebank." *Computational Linguistics* 19 (2): 313–30.

Martens, James. 2010. "Deep Learning via Hessian-Free Optimization." *Proceedings of the 27th International Conference on Machine Learning*, 735–42.

Martens, James, and Roger Grosse. 2015. "Optimizing Neural Networks with Kronecker-Factored Approximate Curvature." *Proceedings of the 32nd International Conference on Machine Learning (ICML)*, 2408–17.

Massias, Mathurin, and Quentin Bertrand. 2026. *Diffusion and Flow-Matching: From Memorization to Generalization & Beyond*. Tutorial at the International Conference on Machine Learning, https://icml.cc/virtual/2026/75374.

McCandlish, Sam, Jared Kaplan, Dario Amodei, and OpenAI Dota Team. 2018. "An Empirical Model of Large-Batch Training." *arXiv Preprint arXiv:1812.06162*.

Mei, Jincheng, Bo Dai, Chenjun Xiao, Csaba Szepesvári, and Dale Schuurmans. 2021. "Understanding the Effect of Stochasticity in Policy Optimization." *Advances in Neural Information Processing Systems* 34.

Mei, Jincheng, Chenjun Xiao, Csaba Szepesvári, and Dale Schuurmans. 2020. "On the Global Convergence Rates of Softmax Policy Gradient Methods." *Proceedings of the 37th*

International Conference on Machine Learning, Proceedings of machine learning research, vol. 119: 6820–29.

Meng, Chenlin, Kristy Choi, Jiaming Song, and Stefano Ermon. 2022. "Concrete Score Matching: Generalized Score Matching for Discrete Data." *Advances in Neural Information Processing Systems* 35.

Meng, Chenlin, Yutong He, Yang Song, et al. 2022. "SDEdit: Guided Image Synthesis and Editing with Stochastic Differential Equations." *International Conference on Learning Representations*. https://arxiv.org/abs/2108.01073.

Meng, Chenlin, Robin Rombach, Ruiqi Gao, et al. 2023. "On Distillation of Guided Diffusion Models." *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 14297–306.

Meng, Kevin, David Bau, Alex Andonian, and Yonatan Belinkov. 2022. "Locating and Editing Factual Associations in GPT." *Advances in Neural Information Processing Systems* 35. https://arxiv.org/abs/2202.05262.

Merity, Stephen, Caiming Xiong, James Bradbury, and Richard Socher. 2016. "Pointer Sentinel Mixture Models." *ArXiv:1609.07843*. https://arxiv.org/abs/1609.07843.

Merrill, William, Shane Arora, Dirk Groeneveld, and Hannaneh Hajishirzi. 2025. "Critical Batch Size Revisited: A Simple Empirical Approach to Large-Batch Language Model Training." *arXiv Preprint arXiv:2505.23971*.

Merrill, William, Jackson Petty, and Ashish Sabharwal. 2024. "The Illusion of State in State-Space Models." *International Conference on Machine Learning*. https://arxiv.org/abs/2404.08819.

Mescheder, Lars, Andreas Geiger, and Sebastian Nowozin. 2018. "Which Training Methods for GANs Do Actually Converge?" *International Conference on Machine Learning*, 3481–90. https://arxiv.org/abs/1801.04406.

Meta AI. 2025. *The Llama 4 Herd: The Beginning of a New Era of Natively Multimodal AI Innovation*. Https://ai.meta.com/blog/llama-4-multimodal-intelligence/.

Metz, Luke, Ben Poole, David Pfau, and Jascha Sohl-Dickstein. 2017. "Unrolled Generative Adversarial Networks." *International Conference on Learning Representations*. https://arxiv.org/abs/1611.02163.

Michel, Paul, Omer Levy, and Graham Neubig. 2019. "Are Sixteen Heads Really Better Than One?" *Advances in Neural Information Processing Systems* 32.

Micikevicius, Paulius, Sharan Narang, Jonah Alben, et al. 2018. "Mixed Precision Training." *International Conference on Learning Representations*.

Micikevicius, Paulius, Dusan Stosic, Neil Burgess, et al. 2022. "FP8 Formats for Deep Learning." *arXiv Preprint arXiv:2209.05433*.

Milakov, Maxim, and Natalia Gimelshein. 2018. "Online Normalizer Calculation for Softmax." *arXiv Preprint arXiv:1805.02867*.

MiniMax. 2025. "MiniMax-01: Scaling Foundation Models with Lightning Attention." *arXiv Preprint arXiv:2501.08313*.

Mirza, Mehdi, and Simon Osindero. 2014. "Conditional Generative Adversarial Nets." *ArXiv:1411.1784*. https://arxiv.org/abs/1411.1784.

Mishchenko, Konstantin, Ahmed Khaled, and Peter Richtárik. 2020. "Random Reshuffling: Simple Analysis with Vast Improvements." *Advances in Neural Information Processing Systems*. https://arxiv.org/abs/2006.05988.

Mistral AI. 2023. *Mistral-7B-Instruct-v0.2 Model Card*. Https://huggingface.co/mistralai/Mistral-7B-Instruct-v0.2.

Miyato, Takeru, Toshiki Kataoka, Masanori Koyama, and Yuichi Yoshida. 2018. "Spectral Normalization for Generative Adversarial Networks." *International Conference on Learning Representations*.

Miyato, Takeru, and Masanori Koyama. 2018. "CGANs with Projection Discriminator." *International Conference on Learning Representations*. https://arxiv.org/abs/1802.05637.

Mnih, Volodymyr, Adrià Puigdomènech Badia, Mehdi Mirza, et al. 2016. "Asynchronous Methods for Deep Reinforcement Learning." *International Conference on Machine Learning*, 1928–37.

Mnih, Volodymyr, Nicolas Heess, Alex Graves, et al. 2014. "Recurrent Models of Visual Attention." *Advances in Neural Information Processing Systems*, 2204–12. https://doi.org/10.5555/2969033.2969073.

Mnih, Volodymyr, Koray Kavukcuoglu, David Silver, et al. 2013. "Playing Atari with Deep Reinforcement Learning." *ArXiv:1312.5602*.

Mnih, Volodymyr, Koray Kavukcuoglu, David Silver, et al. 2015. "Human-Level Control Through Deep Reinforcement Learning." *Nature* 518 (7540): 529–33. https://doi.org/10.1038/nature14236.

Mokady, Ron, Amir Hertz, Kfir Aberman, Yael Pritch, and Daniel Cohen-Or. 2023. "NULL-Text Inversion for Editing Real Images Using Guided Diffusion Models." *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 6038–47.

Muennighoff, Niklas, Alexander M. Rush, Boaz Barak, et al. 2023. "Scaling Data-Constrained Language Models." *Advances in Neural Information Processing Systems*.

Murphy, Kevin P. 2025. "Reinforcement Learning: An Overview." *arXiv Preprint arXiv:2412.05265*.

Nadaraya, Elizbar A. 1964. "On Estimating Regression." *Theory of Probability & Its Applications* 9 (1): 141–42. https://doi.org/10.1137/1109020.

Nado, Zachary, Justin M. Gilmer, Christopher J. Shallue, Rohan Anil, and George E. Dahl. 2021. “A Large Batch Optimizer Reality Check: Traditional, Generic Optimizers Suffice Across Batch Sizes.” *arXiv Preprint arXiv:2102.06356*.

Nagabandi, Anusha, Gregory Kahn, Ronald S. Fearing, and Sergey Levine. 2018. “Neural Network Dynamics for Model-Based Deep Reinforcement Learning with Model-Free Fine-Tuning.” *IEEE International Conference on Robotics and Automation (ICRA)*, 7559–66. https://doi.org/10.1109/ICRA.2018.8463189.

Nakamoto, Mitsuhiko, Yuexiang Zhai, Anikait Singh, et al. 2023. “Cal-QL: Calibrated Offline RL Pre-Training for Efficient Online Fine-Tuning.” *Advances in Neural Information Processing Systems* 36.

Nakkiran, Preetum, Arwen Bradley, Hattie Zhou, and Madhu Advani. 2025. “Step-by-Step Diffusion: An Elementary Tutorial.” *Foundations and Trends in Computer Graphics and Vision* 17 (1): 1–75. https://doi.org/10.1561/0600000114.

Nanda, Neel. 2023. *Attribution Patching: Activation Patching at Industrial Scale*. Blog post, https://www.neelnanda.io/attribution-patching.

Narang, Sharan, Hyung Won Chung, Yi Tay, et al. 2021. “Do Transformer Modifications Transfer Across Implementations and Applications?” *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*. https://arxiv.org/abs/2102.11972.

Narayanan, Deepak, Mohammad Shoeybi, Jared Casper, et al. 2021. “Efficient Large-Scale Language Model Training on GPU Clusters Using Megatron-LM.” *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 1–15. https://doi.org/10.1145/3458817.3476209.

Nauman, Michal, Mateusz Ostaszewski, Krzysztof Jankowski, Piotr Miłoś, and Marek Cygan. 2024. “Bigger, Regularized, Optimistic: Scaling for Compute and Sample-Efficient Continuous Control.” *Advances in Neural Information Processing Systems* 37.

Nemirovski, Arkadi, and David Yudin. 1983. *Problem Complexity and Method Efficiency in Optimization*. Wiley.

Nesterov, Yu. 2018. *Lectures on Convex Optimization*. Springer. https://doi.org/10.1007/978-3-319-91578-4.

Nesterov, Yurii. 1983. “A Method of Solving a Convex Programming Problem with Convergence Rate $O(1/k^2)$.” *Soviet Mathematics Doklady* 27 (2): 372–76.

Ng, Andrew Y, Daishi Harada, and Stuart Russell. 1999. “Policy Invariance Under Reward Transformations: Theory and Application to Reward Shaping.” *International Conference on Machine Learning*, 278–87.

Nguyen, Eric, Michael Poli, Marjan Faizi, et al. 2023. “HyenaDNA: Long-Range Genomic Sequence Modeling at Single Nucleotide Resolution.” *Advances in Neural Information Processing Systems*. https://arxiv.org/abs/2306.15794.

Nguyen, Toan Q., and Julian Salazar. 2019. "Transformers Without Tears: Improving the Normalization of Self-Attention." *Proceedings of the 16th International Conference on Spoken Language Translation*. https://arxiv.org/abs/1910.05895.

Nguyen, XuanLong, Martin J. Wainwright, and Michael I. Jordan. 2009. "On Surrogate Loss Functions and f-Divergences." *The Annals of Statistics* 37 (2): 876–904.

Nichol, Alexander Quinn, and Prafulla Dhariwal. 2021. "Improved Denoising Diffusion Probabilistic Models." *Proceedings of the 38th International Conference on Machine Learning*, Proceedings of machine learning research, vol. 139: 8162–71. https://arxiv.org/abs/2102.09672.

Nie, Shen, Fengqi Zhu, Chao Du, et al. 2025. "Scaling up Masked Diffusion Models on Text." *International Conference on Learning Representations*. https://arxiv.org/abs/2410.18514.

Nie, Shen, Fengqi Zhu, Zebin You, et al. 2025. "Large Language Diffusion Models." *Advances in Neural Information Processing Systems* 38. https://arxiv.org/abs/2502.09992.

Nijkamp, Erik, Mitch Hill, Song-Chun Zhu, and Ying Nian Wu. 2019. "Learning Non-Convergent Non-Persistent Short-Run MCMC Toward Energy-Based Model." *Advances in Neural Information Processing Systems* 32. https://arxiv.org/abs/1904.09770.

Nocedal, Jorge, and Stephen J. Wright. 2006. *Numerical Optimization*. 2nd ed. Springer.

Norelli, Antonio, Marco Fumero, Valentino Maiorca, Luca Moschella, Emanuele Rodolà, and Francesco Locatello. 2022. "ASIF: Coupled Data Turns Unimodal Models to Multimodal Without Training." *ArXiv:2210.01738*. https://arxiv.org/abs/2210.01738.

Nota, Chris, and Philip S. Thomas. 2020. "Is the Policy Gradient a Gradient?" *Proceedings of the 19th International Conference on Autonomous Agents and Multiagent Systems*.

Nowozin, Sebastian, Botond Cseke, and Ryota Tomioka. 2016. "F-GAN: Training Generative Neural Samplers Using Variational Divergence Minimization." *Advances in Neural Information Processing Systems* 29.

Odena, Augustus, Vincent Dumoulin, and Chris Olah. 2016. "Deconvolution and Checkerboard Artifacts." *Distill*. http://distill.pub/2016/deconv-checkerboard.

Odena, Augustus, Christopher Olah, and Jonathon Shlens. 2017. "Conditional Image Synthesis with Auxiliary Classifier GANs." *Proceedings of the 34th International Conference on Machine Learning*, 2642–51. https://arxiv.org/abs/1610.09585.

Olsson, Catherine, Nelson Elhage, Neel Nanda, et al. 2022. "In-Context Learning and Induction Heads." *Transformer Circuits Thread*.

OpenAI. 2025. "Gpt-Oss-120b & Gpt-Oss-20b Model Card." *arXiv Preprint arXiv:2508.10925*.

Oquab, Maxime, Timothée Darcet, Théo Moutakanni, et al. 2024. "DINOv2: Learning Robust Visual Features Without Supervision." *Transactions on Machine Learning Research*. https://arxiv.org/abs/2304.07193.

Orvieto, Antonio, Samuel L. Smith, Albert Gu, et al. 2023. "Resurrecting Recurrent Neural Networks for Long Sequences." *International Conference on Machine Learning*, 26670–98. https://arxiv.org/abs/2303.06349.

Oswald, Johannes von, Eyvind Niklasson, Ettore Randazzo, et al. 2023. "Transformers Learn in-Context by Gradient Descent." *International Conference on Machine Learning*, Proceedings of machine learning research, vol. 202. https://arxiv.org/abs/2212.07677.

Ou, Jingyang, Shen Nie, Kaiwen Xue, et al. 2025. "Your Absorbing Discrete Diffusion Secretly Models the Conditional Distributions of Clean Data." *International Conference on Learning Representations*. https://arxiv.org/abs/2406.03736.

Ouyang, Long, Jeff Wu, Xu Jiang, et al. 2022. "Training Language Models to Follow Instructions with Human Feedback." *ArXiv:2203.02155*. https://arxiv.org/abs/2203.02155.

Pardo, Fabio, Arash Tavakoli, Vitaly Levdik, and Petar Kormushev. 2018. "Time Limits in Reinforcement Learning." *International Conference on Machine Learning*, 4045–54. https://arxiv.org/abs/1712.00378.

Park, Taesung, Ming-Yu Liu, Ting-Chun Wang, and Jun-Yan Zhu. 2019. "Semantic Image Synthesis with Spatially-Adaptive Normalization." *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2337–46. https://doi.org/10.1109/cvpr.2019.00244.

Parmar, Gaurav, Richard Zhang, and Jun-Yan Zhu. 2022. "On Aliased Resizing and Surprising Subtleties in GAN Evaluation." *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 11410–20. https://arxiv.org/abs/2104.11222.

Parzen, Emanuel. 1957. "On Consistent Estimates of the Spectrum of a Stationary Time Series." *Annals of Mathematical Statistics* 28: 329–48. https://doi.org/10.1214/aoms/1177706962.

Paster, Keiran, Sheila A. McIlraith, and Jimmy Ba. 2022. "You Can't Count on Luck: Why Decision Transformers and RvS Fail in Stochastic Environments." *Advances in Neural Information Processing Systems* 35.

Patarasuk, Pitch, and Xin Yuan. 2009. "Bandwidth Optimal All-Reduce Algorithms for Clusters of Workstations." *Journal of Parallel and Distributed Computing* 69 (2): 117–24. https://doi.org/10.1016/j.jpdc.2008.09.002.

Pearce, Tim, and Jinyeop Song. 2024. "Reconciling Kaplan and Chinchilla Scaling Laws." *Transactions on Machine Learning Research*. https://arxiv.org/abs/2406.12907.

Peebles, William, and Saining Xie. 2023. "Scalable Diffusion Models with Transformers." *Proceedings of the IEEE/CVF International Conference on Computer Vision*, 4195–205. https://arxiv.org/abs/2212.09748.

Peng, Bowen, Jeffrey Quesnelle, Honglu Fan, and Enrico Shippole. 2024. "YaRN: Efficient Context Window Extension of Large Language Models." *International Conference on Learning Representations*. https://arxiv.org/abs/2309.00071.

Peng, Bo, Ruichong Zhang, Daniel Goldstein, et al. 2025. "RWKV-7 "Goose" with Expressive Dynamic State Evolution." *arXiv Preprint arXiv:2503.14456*.

Perez, Ethan, Florian Strub, Harm de Vries, Vincent Dumoulin, and Aaron Courville. 2018. "FiLM: Visual Reasoning with a General Conditioning Layer." *Proceedings of the AAAI Conference on Artificial Intelligence* 32. https://arxiv.org/abs/1709.07871.

Pethick, Thomas, Wanyun Xie, Kimon Antonakopoulos, Zhenyu Zhu, Antonio Silveti-Falls, and Volkan Cevher. 2025. "Training Deep Learning Models with Norm-Constrained LMOs." *International Conference on Machine Learning*.

Podell, Dustin, Zion English, Kyle Lacey, et al. 2024. "SDXL: Improving Latent Diffusion Models for High-Resolution Image Synthesis." *International Conference on Learning Representations*. https://arxiv.org/abs/2307.01952.

Poli, Michael, Stefano Massaroli, Eric Nguyen, et al. 2023. "Hyena Hierarchy: Towards Larger Convolutional Language Models." *International Conference on Machine Learning*. https://arxiv.org/abs/2302.10866.

Polyak, Adam, Amit Zohar, Andrew Brown, et al. 2024. "Movie Gen: A Cast of Media Foundation Models." *arXiv Preprint arXiv:2410.13720*. https://arxiv.org/abs/2410.13720.

Polyak, Boris T. 1964. "Some Methods of Speeding up the Convergence of Iteration Methods." *USSR Computational Mathematics and Mathematical Physics* 4 (5): 1–17. https://doi.org/10.1016/0041-5553(64)90137-5.

Polyak, Boris T., and Anatoli B. Juditsky. 1992. "Acceleration of Stochastic Approximation by Averaging." *SIAM Journal on Control and Optimization* 30 (4): 838–55.

Pomerleau, Dean A. 1989. "ALVINN: An Autonomous Land Vehicle in a Neural Network." In *Advances in Neural Information Processing Systems 1*, edited by David S Touretzky. Morgan Kaufmann.

Pomerleau, Dean A. 1991. "Efficient Training of Artificial Neural Networks for Autonomous Navigation." *Neural Computation* 3 (1): 88–97.

Pope, Reiner, Sholto Douglas, Aakanksha Chowdhery, et al. 2023. "Efficiently Scaling Transformer Inference." *Proceedings of Machine Learning and Systems* 5.

Porian, Tomer, Mitchell Wortsman, Jenia Jitsev, Ludwig Schmidt, and Yair Carmon. 2024. "Resolving Discrepancies in Compute-Optimal Scaling of Language Models." *Advances in Neural Information Processing Systems* 37. https://arxiv.org/abs/2406.19146.

Prabhudesai, Mihir, Mengning Wu, Amir Zadeh, Katerina Fragkiadaki, and Deepak Pathak. 2025. "Diffusion Beats Autoregressive in Data-Constrained Settings." *arXiv Preprint arXiv:2507.15857*. https://arxiv.org/abs/2507.15857.

Press, Ofir, Noah A. Smith, and Mike Lewis. 2022. "Train Short, Test Long: Attention with Linear Biases Enables Input Length Extrapolation." *International Conference on Learning Representations*. https://arxiv.org/abs/2108.12409.

Press, Ofir, and Lior Wolf. 2017. "Using the Output Embedding to Improve Language Models." *Proceedings of the 15th Conference of the European Chapter of the Association for Computational Linguistics*, 157–63.

Puterman, Martin L. 1994. *Markov Decision Processes: Discrete Stochastic Dynamic Programming*. John Wiley & Sons. https://doi.org/10.1002/9780470316887.

Qin, Zhen, Weixuan Sun, Hui Deng, et al. 2022. "cosFormer: Rethinking Softmax in Attention." *International Conference on Learning Representations*. https://arxiv.org/abs/2202.08791.

Qiu, Zihan, Zekun Wang, Bo Zheng, et al. 2025. "Gated Attention for Large Language Models: Non-Linearity, Sparsity, and Attention-Sink-Free." *Advances in Neural Information Processing Systems*. https://arxiv.org/abs/2505.06708.

Qwen Team. 2025. *Qwen3-Next: Towards Ultimate Training and Inference Efficiency*. Model release, https://huggingface.co/Qwen/Qwen3-Next-80B-A3B-Instruct.

Rabe, Markus N., and Charles Staats. 2021. "Self-Attention Does Not Need $O(n^2)$ Memory." *arXiv Preprint arXiv:2112.05682*.

Rabiner, Lawrence, and Biing-Hwang Juang. 1993. *Fundamentals of Speech Recognition*. Prentice-Hall.

Radford, Alec, Jong Wook Kim, Chris Hallacy, et al. 2021. "Learning Transferable Visual Models from Natural Language Supervision." *International Conference on Machine Learning*, 8748–63. https://proceedings.mlr.press/v139/radford21a.html.

Radford, Alec, Jong Wook Kim, Tao Xu, Greg Brockman, Christine McLeavey, and Ilya Sutskever. 2023. "Robust Speech Recognition via Large-Scale Weak Supervision." *International Conference on Machine Learning*. https://arxiv.org/abs/2212.04356.

Radford, Alec, Luke Metz, and Soumith Chintala. 2015. "Unsupervised Representation Learning with Deep Convolutional Generative Adversarial Networks." *ArXiv:1511.06434*. https://arxiv.org/abs/1511.06434.

Radford, Alec, Karthik Narasimhan, Tim Salimans, and Ilya Sutskever. 2018. "Improving Language Understanding by Generative Pre-Training." *OpenAI*. https://cdn.openai.com/research-covers/language-unsupervised/language_understanding_paper.pdf.

Radford, Alec, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, and Ilya Sutskever. 2019. "Language Models Are Unsupervised Multitask Learners." *OpenAI Blog* 1 (8): 9. https://cdn.openai.com/better-language-models/language_models_are_unsupervised_multitask_learners.pdf.

Rafailov, Rafael, Archit Sharma, Eric Mitchell, Stefano Ermon, Christopher D Manning, and Chelsea Finn. 2023. "Direct Preference Optimization: Your Language Model Is Secretly a Reward Model." *Advances in Neural Information Processing Systems* 36.

Raffel, Colin, Noam Shazeer, Adam Roberts, et al. 2020. "Exploring the Limits of Transfer

Learning with a Unified Text-to-Text Transformer." *Journal of Machine Learning Research* 21: 1–67. https://arxiv.org/abs/1910.10683.

Raffin, Antonin, Ashley Hill, Adam Gleave, Anssi Kanervisto, Maximilian Ernestus, and Noah Dormann. 2021. "Stable-Baselines3: Reliable Reinforcement Learning Implementations." *Journal of Machine Learning Research* 22 (268): 1–8.

Rajbhandari, Samyam, Jeff Rasley, Olatunji Ruwase, and Yuxiong He. 2020. "ZeRO: Memory Optimizations Toward Training Trillion Parameter Models." *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*, 1–16.

Rajbhandari, Samyam, Olatunji Ruwase, Jeff Rasley, Shaden Smith, and Yuxiong He. 2021. "ZeRO-Infinity: Breaking the GPU Memory Wall for Extreme Scale Deep Learning." *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. https://arxiv.org/abs/2104.07857.

Ramachandran, Prajit, Niki Parmar, Ashish Vaswani, Irwan Bello, Anselm Levskaya, and Jon Shlens. 2019. "Stand-Alone Self-Attention in Vision Models." *Advances in Neural Information Processing Systems* 32. https://arxiv.org/abs/1906.05909.

Ramapuram, Jason, Federico Danieli, Eeshan Dhekane, et al. 2025. "Theory, Analysis, and Best Practices for Sigmoid Self-Attention." *International Conference on Learning Representations*. https://arxiv.org/abs/2409.04431.

Ramsauer, Hubert, Bernhard Schäfl, Johannes Lehner, et al. 2021. "Hopfield Networks Is All You Need." *International Conference on Learning Representations*. https://arxiv.org/abs/2008.02217.

Randløv, Jette, and Preben Alstrøm. 1998. "Learning to Drive a Bicycle Using Reinforcement Learning and Shaping." *Proceedings of the Fifteenth International Conference on Machine Learning*, 463–71.

Rashidinejad, Paria, Banghua Zhu, Cong Ma, Jiantao Jiao, and Stuart Russell. 2021. "Bridging Offline Reinforcement Learning and Imitation Learning: A Tale of Pessimism." *Advances in Neural Information Processing Systems* 34.

Ravuri, Suman, and Oriol Vinyals. 2019. "Classification Accuracy Score for Conditional Generative Models." *Advances in Neural Information Processing Systems* 32. https://arxiv.org/abs/1905.10887.

Reddi, Sashank J, Satyen Kale, and Sanjiv Kumar. 2018. "On the Convergence of Adam and Beyond." *International Conference on Learning Representations*. https://arxiv.org/abs/1904.09237.

Ren, Jie, Samyam Rajbhandari, Reza Yazdani Aminabadi, et al. 2021. "ZeRO-Offload: Democratizing Billion-Scale Model Training." *2021 USENIX Annual Technical Conference (USENIX ATC 21)*, 551–64. https://www.usenix.org/conference/atc21/presentation/ren-jie.

Ren, Liliang, Yang Liu, Yadong Lu, Yelong Shen, Chen Liang, and Weizhu Chen. 2024.

"Samba: Simple Hybrid State Space Models for Efficient Unlimited Context Language Modeling." *arXiv Preprint arXiv:2406.07522*.

Robbins, Herbert. 1952. "Some Aspects of the Sequential Design of Experiments." *Bulletin of the American Mathematical Society* 58 (5): 527–35. https://doi.org/10.1090/S0002-9904-1952-09620-8.

Robbins, Herbert. 1956. "An Empirical Bayes Approach to Statistics." *Proceedings of the Third Berkeley Symposium on Mathematical Statistics and Probability, Volume 1*, 157–63.

Robbins, Herbert, and Sutton Monro. 1951. "A Stochastic Approximation Method." *The Annals of Mathematical Statistics* 22 (3): 400–407.

Roberts, Gareth O., and Richard L. Tweedie. 1996. "Exponential Convergence of Langevin Distributions and Their Discrete Approximations." *Bernoulli* 2 (4): 341–63.

Rombach, Robin, Andreas Blattmann, Dominik Lorenz, Patrick Esser, and Björn Ommer. 2022. "High-Resolution Image Synthesis with Latent Diffusion Models." *IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 10684–95.

Ronneberger, Olaf, Philipp Fischer, and Thomas Brox. 2015. "U-Net: Convolutional Networks for Biomedical Image Segmentation." *Medical Image Computing and Computer-Assisted Intervention (MICCAI)*, Lecture notes in computer science, vol. 9351: 234–41. https://arxiv.org/abs/1505.04597.

Ross, Stéphane, and J Andrew Bagnell. 2010. "Efficient Reductions for Imitation Learning." *Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics*, Proceedings of machine learning research, vol. 9: 661–68.

Ross, Stéphane, Geoffrey J Gordon, and J Andrew Bagnell. 2011. "A Reduction of Imitation Learning and Structured Prediction to No-Regret Online Learning." *International Conference on Artificial Intelligence and Statistics*, 627–35.

Roth, Kevin, Aurelien Lucchi, Sebastian Nowozin, and Thomas Hofmann. 2017. "Stabilizing Training of Generative Adversarial Networks Through Regularization." *Advances in Neural Information Processing Systems* 30. https://arxiv.org/abs/1705.09367.

Rouhani, Bita Darvish, Ritchie Zhao, Ankit More, et al. 2023. "Microscaling Data Formats for Deep Learning." *arXiv Preprint arXiv:2310.10537*.

Rubinstein, Reuven Y., and Dirk P. Kroese. 2004. *The Cross-Entropy Method: A Unified Approach to Combinatorial Optimization, Monte-Carlo Simulation, and Machine Learning*. Springer. https://doi.org/10.1007/978-1-4757-4321-0.

Rummery, Gavin A, and Mahesan Niranjan. 1994. *On-Line Q-Learning Using Connectionist Systems*. CUED/F-INFENG/TR 166. Cambridge University Engineering Department.

Sadat, Seyedmorteza, Otmar Hilliges, and Romann M. Weber. 2025. "Eliminating Oversaturation and Artifacts of High Guidance Scales in Diffusion Models." *International Conference on Learning Representations*. https://arxiv.org/abs/2410.02416.

Saharia, Chitwan, William Chan, Saurabh Saxena, et al. 2022. "Photorealistic Text-to-Image Diffusion Models with Deep Language Understanding." *ArXiv:2205.11487*. https://arxiv.org/abs/2205.11487.

Sahoo, Subham Sekhar, Marianne Arriola, Yair Schiff, et al. 2024. "Simple and Effective Masked Diffusion Language Models." *Advances in Neural Information Processing Systems* 37. https://arxiv.org/abs/2406.07524.

Sahoo, Subham Sekhar, Justin Deschenaux, Aaron Gokaslan, Guanghan Wang, Justin Chiu, and Volodymyr Kuleshov. 2025. "The Diffusion Duality." *International Conference on Machine Learning*.

Salimans, Tim, and Jonathan Ho. 2022. "Progressive Distillation for Fast Sampling of Diffusion Models." *International Conference on Learning Representations*.

Salimans, Tim, Jonathan Ho, Xi Chen, Szymon Sidor, and Ilya Sutskever. 2017. "Evolution Strategies as a Scalable Alternative to Reinforcement Learning." *arXiv Preprint arXiv:1703.03864*.

Samuel, Arthur L. 1959. "Some Studies in Machine Learning Using the Game of Checkers." *IBM Journal of Research and Development* 3 (3): 210–29. https://doi.org/10.1147/rd.33.0210.

Santurkar, Shibani, Dimitris Tsipras, Andrew Ilyas, and Aleksander Madry. 2018. "How Does Batch Normalization Help Optimization?" *Advances in Neural Information Processing Systems*, 2483–93. https://doi.org/10.5555/3327345.3327508.

Sardana, Nikhil, Jacob Portes, Sasha Doubov, and Jonathan Frankle. 2024. "Beyond Chinchilla-Optimal: Accounting for Inference in Language Model Scaling Laws." *International Conference on Machine Learning*.

Sauer, Axel, Frederic Boesel, Tim Dockhorn, Andreas Blattmann, Patrick Esser, and Robin Rombach. 2024. "Fast High-Resolution Image Synthesis with Latent Adversarial Diffusion Distillation." *SIGGRAPH Asia 2024 Conference Papers*.

Sauer, Axel, Kashyap Chitta, Jens Müller, and Andreas Geiger. 2021. "Projected GANs Converge Faster." *Advances in Neural Information Processing Systems* 34.

Sauer, Axel, Tero Karras, Samuli Laine, Andreas Geiger, and Timo Aila. 2023. "StyleGAN-T: Unlocking the Power of GANs for Fast Large-Scale Text-to-Image Synthesis." *Proceedings of the 40th International Conference on Machine Learning*.

Sauer, Axel, Dominik Lorenz, Andreas Blattmann, and Robin Rombach. 2023. "Adversarial Diffusion Distillation." *arXiv Preprint arXiv:2311.17042*.

Schaul, Tom, John Quan, Ioannis Antonoglou, and David Silver. 2016. "Prioritized Experience Replay." *International Conference on Learning Representations*. https://arxiv.org/abs/1511.05952.

Schiff, Yair, Chia-Hsiang Kao, Aaron Gokaslan, Tri Dao, Albert Gu, and Volodymyr Kuleshov.

2024. “Caduceus: Bi-Directional Equivariant Long-Range DNA Sequence Modeling.” *International Conference on Machine Learning*. https://arxiv.org/abs/2403.03234.

Schiff, Yair, Subham Sekhar Sahoo, Hao Phung, et al. 2025. “Simple Guidance Mechanisms for Discrete Diffusion Models.” *International Conference on Learning Representations*.

Schlag, Imanol, Kazuki Irie, and Jürgen Schmidhuber. 2021. “Linear Transformers Are Secretly Fast Weight Programmers.” *International Conference on Machine Learning*.

Schmidt, Robin M., Frank Schneider, and Philipp Hennig. 2021. “Descending Through a Crowded Valley — Benchmarking Deep Learning Optimizers.” *Proceedings of the 38th International Conference on Machine Learning*, 9367–76. https://arxiv.org/abs/2007.01547.

Schrittwieser, Julian, Ioannis Antonoglou, Thomas Hubert, et al. 2020. “Mastering Atari, Go, Chess and Shogi by Planning with a Learned Model.” *Nature* 588 (7839): 604–9. https://doi.org/10.1038/s41586-020-03051-0.

Schulman, John, Sergey Levine, Pieter Abbeel, Michael Jordan, and Philipp Moritz. 2015. “Trust Region Policy Optimization.” *International Conference on Machine Learning*, 1889–97.

Schulman, John, Philipp Moritz, Sergey Levine, Michael Jordan, and Pieter Abbeel. 2016. “High-Dimensional Continuous Control Using Generalized Advantage Estimation.” *International Conference on Learning Representations*.

Schulman, John, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. 2017. “Proximal Policy Optimization Algorithms.” *arXiv Preprint arXiv:1707.06347*.

Schuster, Mike, and Kuldip K Paliwal. 1997. “Bidirectional Recurrent Neural Networks.” *IEEE Transactions on Signal Processing* 45 (11): 2673–81. https://doi.org/10.1109/78.650093.

Semenov, Andrei, Matteo Pagliardini, and Martin Jaggi. 2025. “Benchmarking Optimizers for Large Language Model Pretraining.” *arXiv Preprint arXiv:2509.01440*.

Shah, Ishaan, Anthony M. Polloreno, Karl Stratos, et al. 2025. “Practical Efficiency of Muon for Pretraining.” *arXiv Preprint arXiv:2505.02222*.

Shah, Jay, Ganesh Bikshandi, Ying Zhang, Vijay Thakkar, Pradeep Ramani, and Tri Dao. 2024. “FlashAttention-3: Fast and Accurate Attention with Asynchrony and Low-Precision.” *Advances in Neural Information Processing Systems* 37. https://arxiv.org/abs/2407.08608.

Shallue, Christopher J., Jaehoon Lee, Joseph Antognini, Jascha Sohl-Dickstein, Roy Frostig, and George E. Dahl. 2019. “Measuring the Effects of Data Parallelism on Neural Network Training.” *Journal of Machine Learning Research* 20 (112): 1–49. https://arxiv.org/abs/1811.03600.

Shao, Zhihong, Peiyi Wang, Qihao Zhu, et al. 2024. “DeepSeekMath: Pushing the Limits of Mathematical Reasoning in Open Language Models.” *arXiv Preprint arXiv:2402.03300*.

Shaw, Peter, Jakob Uszkoreit, and Ashish Vaswani. 2018. "Self-Attention with Relative Position Representations." *ArXiv:1803.02155*. https://arxiv.org/abs/1803.02155.

Shazeer, Noam. 2019. "Fast Transformer Decoding: One Write-Head Is All You Need." *arXiv Preprint arXiv:1911.02150*.

Shazeer, Noam. 2020. "GLU Variants Improve Transformer." *ArXiv:2002.05202*. https://arxiv.org/abs/2002.05202.

Shazeer, Noam, Azalia Mirhoseini, Krzysztof Maziarz, et al. 2017. "Outrageously Large Neural Networks: The Sparsely-Gated Mixture-of-Experts Layer." *International Conference on Learning Representations*.

Shazeer, Noam, and Mitchell Stern. 2018. "Adafactor: Adaptive Learning Rates with Sublinear Memory Cost." *International Conference on Machine Learning*. https://arxiv.org/abs/1804.04235.

Shen, Lingfeng, Aayush Mishra, and Daniel Khashabi. 2024. "Position: Do Pretrained Transformers Learn in-Context by Gradient Descent?" *International Conference on Machine Learning*, Proceedings of machine learning research, vol. 235: 44712–40. https://arxiv.org/abs/2310.08540.

Shi, Jiaxin, Kehang Han, Zhe Wang, Arnaud Doucet, and Michalis K. Titsias. 2024. "Simplified and Generalized Masked Diffusion for Discrete Data." *Advances in Neural Information Processing Systems* 37. https://arxiv.org/abs/2406.04329.

Shih, Andy, Suneel Belkhale, Stefano Ermon, Dorsa Sadigh, and Nima Anari. 2023. "Parallel Sampling of Diffusion Models." *Advances in Neural Information Processing Systems* 36. https://arxiv.org/abs/2305.16317.

Shoeybi, Mohammad, Mostofa Patwary, Raul Puri, Patrick LeGresley, Jared Casper, and Bryan Catanzaro. 2019. "Megatron-LM: Training Multi-Billion Parameter Language Models Using Model Parallelism." *ArXiv:1909.08053*. https://arxiv.org/abs/1909.08053.

Si, Chongjie, Debing Zhang, and Wei Shen. 2025. "AdaMuon: Adaptive Muon Optimizer." *arXiv Preprint arXiv:2507.11005*.

Siems, Julien, Timur Carstensen, Arber Zela, Frank Hutter, Massimiliano Pontil, and Riccardo Grazzi. 2025. "DeltaProduct: Improving State-Tracking in Linear RNNs via Householder Products." *arXiv Preprint arXiv:2502.10297*.

Silver, David, Aja Huang, Chris J Maddison, et al. 2016. "Mastering the Game of Go with Deep Neural Networks and Tree Search." *Nature* 529 (7587): 484–89. https://doi.org/10.1038/nature16961.

Silver, David, Thomas Hubert, Julian Schrittwieser, et al. 2018. "A General Reinforcement Learning Algorithm That Masters Chess, Shogi, and Go Through Self-Play." *Science* 362 (6419): 1140–44. https://doi.org/10.1126/science.aar6404.

Silver, David, Guy Lever, Nicolas Heess, Thomas Degris, Daan Wierstra, and Martin Ried-

miller. 2014. "Deterministic Policy Gradient Algorithms." *Proceedings of the 31st International Conference on Machine Learning*, Proceedings of machine learning research, vol. 32: 387–95. https://proceedings.mlr.press/v32/silver14.html.

Silver, David, Julian Schrittwieser, Karen Simonyan, et al. 2017. "Mastering the Game of Go Without Human Knowledge." *Nature* 550 (7676): 354–59. https://doi.org/10.1038/nature24270.

Silverman, B. W. 1986. *Density Estimation for Statistical and Data Analysis*. Chapman; Hall.

Singh, Satinder P, and Richard C Yee. 1994. "An Upper Bound on the Loss from Approximate Optimal-Value Functions." *Machine Learning* 16 (3): 227–33. https://doi.org/10.1007/BF00993308.

Singhania, Prajwal, Siddharth Singh, Shwai He, Soheil Feizi, and Abhinav Bhatele. 2024. "Loki: Low-Rank Keys for Efficient Sparse Attention." *Advances in Neural Information Processing Systems* 37. https://arxiv.org/abs/2406.02542.

Skalse, Joar, Matthew Farrugia-Roberts, Stuart Russell, Alessandro Abate, and Adam Gleave. 2023. "Invariance in Policy Optimisation and Partial Identifiability in Reward Learning." *International Conference on Machine Learning*. https://arxiv.org/abs/2203.07475.

Smith, Jimmy T. H., Andrew Warrington, and Scott W. Linderman. 2023. "Simplified State Space Layers for Sequence Modeling." *International Conference on Learning Representations*. https://arxiv.org/abs/2208.04933.

Smith, Samuel L., Pieter-Jan Kindermans, Chris Ying, and Quoc V. Le. 2018. "Don't Decay the Learning Rate, Increase the Batch Size." *International Conference on Learning Representations*. https://arxiv.org/abs/1711.00489.

Sohl-Dickstein, Jascha, Eric Weiss, Niru Maheswaranathan, and Surya Ganguli. 2015. "Deep Unsupervised Learning Using Nonequilibrium Thermodynamics." *International Conference on Machine Learning*, 2256–65. https://proceedings.mlr.press/v37/sohl-dickstein15.html.

Somepalli, Gowthami, Vasu Singla, Micah Goldblum, Jonas Geiping, and Tom Goldstein. 2023. "Diffusion Art or Digital Forgery? Investigating Data Replication in Diffusion Models." *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*. https://arxiv.org/abs/2212.03860.

Song, Jiaming, Chenlin Meng, and Stefano Ermon. 2021. "Denoising Diffusion Implicit Models." *International Conference on Learning Representations*.

Song, Yang. 2021. *Generative Modeling by Estimating Gradients of the Data Distribution*. Blog post, https://yang-song.net/blog/2021/score/.

Song, Yang, and Prafulla Dhariwal. 2024. "Improved Techniques for Training Consistency Models." *International Conference on Learning Representations*. https://arxiv.org/abs/2310.14189.

Song, Yang, Prafulla Dhariwal, Mark Chen, and Ilya Sutskever. 2023. "Consistency Models." *Proceedings of the 40th International Conference on Machine Learning*.

Song, Yang, Conor Durkan, Iain Murray, and Stefano Ermon. 2021. "Maximum Likelihood Training of Score-Based Diffusion Models." *Advances in Neural Information Processing Systems* 34.

Song, Yang, and Stefano Ermon. 2019. "Generative Modeling by Estimating Gradients of the Data Distribution." *Advances in Neural Information Processing Systems* 32. https://arxiv.org/abs/1907.05600.

Song, Yang, and Stefano Ermon. 2020. "Improved Techniques for Training Score-Based Generative Models." *Advances in Neural Information Processing Systems* 33. https://arxiv.org/abs/2006.09011.

Song, Yang, and Diederik P. Kingma. 2021. "How to Train Your Energy-Based Models." *arXiv Preprint arXiv:2101.03288*. https://arxiv.org/abs/2101.03288.

Song, Yang, Jascha Sohl-Dickstein, Diederik P Kingma, Abhishek Kumar, Stefano Ermon, and Ben Poole. 2021. "Score-Based Generative Modeling Through Stochastic Differential Equations." *International Conference on Learning Representations*. https://arxiv.org/abs/2011.13456.

Sriperumbudur, Bharath K., Kenji Fukumizu, Arthur Gretton, Bernhard Schölkopf, and Gert R. G. Lanckriet. 2009. "On Integral Probability Metrics, ϕ-Divergences and Binary Classification." *arXiv Preprint arXiv:0901.2698*.

Sriperumbudur, Bharath K., Kenji Fukumizu, Arthur Gretton, Bernhard Schölkopf, and Gert R. G. Lanckriet. 2012. "On the Empirical Estimation of Integral Probability Metrics." *Electronic Journal of Statistics* 6: 1550–99.

Stein, George, Jesse C. Cresswell, Rasa Hosseinzadeh, et al. 2023. "Exposing Flaws of Generative Model Evaluation Metrics and Their Unfair Treatment of Diffusion Models." *Advances in Neural Information Processing Systems* 36. https://arxiv.org/abs/2306.04675.

Steiner, Andreas, Alexander Kolesnikov, Xiaohua Zhai, Ross Wightman, Jakob Uszkoreit, and Lucas Beyer. 2022. "How to Train Your ViT? Data, Augmentation, and Regularization in Vision Transformers." *Transactions on Machine Learning Research*. https://arxiv.org/abs/2106.10270.

Strehl, Alexander L., and Michael L. Littman. 2008. "An Analysis of Model-Based Interval Estimation for Markov Decision Processes." *Journal of Computer and System Sciences* 74 (8): 1309–31. https://doi.org/10.1016/j.jcss.2007.08.009.

Su, Jianlin, Yu Lu, Shengfeng Pan, Ahmed Murtadha, Bo Wen, and Yunfeng Liu. 2024. "RoFormer: Enhanced Transformer with Rotary Position Embedding." *Neurocomputing* 568: 127063. https://arxiv.org/abs/2104.09864.

Sun, Mingjie, Xinlei Chen, J. Zico Kolter, and Zhuang Liu. 2024. "Massive Activations in Large Language Models." *Conference on Language Modeling (COLM)*. https://arxiv.org/ab

s/2402.17762.

Sun, Ruoyu, Tiantian Fang, and Alexander Schwing. 2020. "Towards a Better Global Loss Landscape of GANs." *Advances in Neural Information Processing Systems* 33.

Sun, Yu, Xinhao Li, Karan Dalal, et al. 2024. "Learning to (Learn at Test Time): RNNs with Expressive Hidden States." *arXiv Preprint arXiv:2407.04620*.

Sun, Yutao, Li Dong, Shaohan Huang, et al. 2023. "Retentive Network: A Successor to Transformer for Large Language Models." *arXiv Preprint arXiv:2307.08621*.

Sutskever, Ilya, James Martens, George Dahl, and Geoffrey Hinton. 2013. "On the Importance of Initialization and Momentum in Deep Learning." *International Conference on Machine Learning*, 1139–47. https://proceedings.mlr.press/v28/sutskever13.html.

Sutton, Richard S. 1988. "Learning to Predict by the Methods of Temporal Differences." *Machine Learning* 3 (1): 9–44. https://doi.org/10.1007/BF00115009.

Sutton, Richard S. 1990. "Integrated Architectures for Learning, Planning, and Reacting Based on Approximating Dynamic Programming." *Proceedings of the Seventh International Conference on Machine Learning*, 216–24.

Sutton, Richard S, and Andrew G Barto. 2018. *Reinforcement Learning: An Introduction*. 2nd ed. MIT Press. http://incompleteideas.net/book/RLbook2020.pdf.

Sutton, Richard S, David McAllester, Satinder Singh, and Yishay Mansour. 2000. "Policy Gradient Methods for Reinforcement Learning with Function Approximation." *Advances in Neural Information Processing Systems* 12.

Szepesvári, Csaba. 2010. *Algorithms for Reinforcement Learning*. Synthesis Lectures on Artificial Intelligence and Machine Learning 9. Morgan; Claypool. https://sites.ualberta.ca/~szepesva/papers/RLAlgsInMDPs.pdf.

Tallec, Corentin, and Yann Ollivier. 2018. "Can Recurrent Neural Networks Warp Time?" *International Conference on Learning Representations*. https://arxiv.org/abs/1804.11188.

Tang, Yunhao, and Rémi Munos. 2025. "On a Few Pitfalls in KL Divergence Gradient Estimation for RL." *arXiv Preprint arXiv:2506.09477*.

Tay, Yi, Mostafa Dehghani, Dara Bahri, and Donald Metzler. 2020. "Efficient Transformers: A Survey." *ArXiv:2009.06732*. https://arxiv.org/abs/2009.06732.

Team OLMo. 2025. "Olmo 3." *arXiv Preprint arXiv:2512.13961*.

Team OLMo, Pete Walsh, Luca Soldaini, et al. 2025. "2 OLMo 2 Furious." *arXiv Preprint arXiv:2501.00656*.

Tesauro, Gerald. 1995. "Temporal Difference Learning and TD-Gammon." *Communications of the ACM* 38 (3): 58–68. https://doi.org/10.1145/203330.203343.

Thakur, Rajeev, Rolf Rabenseifner, and William Gropp. 2005. "Optimization of Collective Communication Operations in MPICH." *The International Journal of High Performance Computing Applications* 19 (1): 49–66. https://doi.org/10.1177/1094342005051521.

Theis, Lucas, Aäron van den Oord, and Matthias Bethge. 2016. "A Note on the Evaluation of Generative Models." *International Conference on Learning Representations*.

Thomas, Philip. 2014. "Bias in Natural Actor-Critic Algorithms." In *Proceedings of the 31st International Conference on Machine Learning*, edited by Eric P. Xing and Tony Jebara, vol. 32. Proceedings of Machine Learning Research 1.

Thompson, William R. 1933. "On the Likelihood That One Unknown Probability Exceeds Another in View of the Evidence of Two Samples." *Biometrika* 25 (3/4): 285–94.

Thrun, Sebastian, and Anton Schwartz. 1993. "Issues in Using Function Approximation for Reinforcement Learning." *Proceedings of the 4th Connectionist Models Summer School*, 255–63.

Tieleman, Tijmen. 2008. "Training Restricted Boltzmann Machines Using Approximations to the Likelihood Gradient." *Proceedings of the 25th International Conference on Machine Learning*, 1064–71.

Tieleman, Tijmen, and Geoffrey Hinton. 2012. "Divide the Gradient by a Running Average of Its Recent Magnitude." In *COURSERA: Neural Networks for Machine Learning, Lecture 6.5-Rmsprop*. https://www.cs.toronto.edu/~tijmen/csc321/slides/lecture_slides_lec6.pdf.

Tillet, Philippe, H. T. Kung, and David Cox. 2019. "Triton: An Intermediate Language and Compiler for Tiled Neural Network Computations." *3rd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages (MAPL)*, 10–19. https://doi.org/10.1145/3315508.3329973.

Touvron, Hugo, Matthieu Cord, Matthijs Douze, Francisco Massa, Alexandre Sablayrolles, and Hervé Jégou. 2021. "Training Data-Efficient Image Transformers & Distillation Through Attention." *International Conference on Machine Learning*, 10347–57. https://proceedings.mlr.press/v139/touvron21a.html.

Touvron, Hugo, Thibaut Lavril, Gautier Izacard, et al. 2023a. "LLaMA: Open and Efficient Foundation Language Models." *ArXiv:2302.13971*, 2023a. https://arxiv.org/abs/2302.13971.

Touvron, Hugo, Louis Martin, Kevin Stone, et al. 2023b. "LLaMA 2: Open Foundation and Fine-Tuned Chat Models." *ArXiv:2307.09288*, 2023b. https://arxiv.org/abs/2307.09288.

Towers, Mark, Ariel Kwiatkowski, Jordan Terry, et al. 2024. "Gymnasium: A Standard Interface for Reinforcement Learning Environments." *arXiv Preprint arXiv:2407.17032*.

Tsitsiklis, John N. 1994. "Asynchronous Stochastic Approximation and Q-Learning." *Machine Learning* 16: 185–202. https://doi.org/10.1007/BF00993306.

Tsitsiklis, John N, and Benjamin Van Roy. 1997. "An Analysis of Temporal-Difference Learn-

ing with Function Approximation." *IEEE Transactions on Automatic Control* 42 (5): 674–90. https://doi.org/10.1109/9.580874.

Uria, Benigno, Iain Murray, and Hugo Larochelle. 2014. "A Deep and Tractable Density Estimator." *Proceedings of the 31st International Conference on Machine Learning*, Proceedings of machine learning research, vol. 32: 467–75.

Van Hasselt, Hado, Arthur Guez, and David Silver. 2016. "Deep Reinforcement Learning with Double Q-Learning." *Proceedings of the AAAI Conference on Artificial Intelligence* 30.

Vaswani, Ashish, Noam Shazeer, Niki Parmar, et al. 2017. "Attention Is All You Need." *Advances in Neural Information Processing Systems*, 5998–6008. https://doi.org/10.5555/3295222.3295349.

Vincent, Pascal. 2011. "A Connection Between Score Matching and Denoising Autoencoders." *Neural Computation* 23 (7): 1661–74.

Voita, Elena, David Talbot, Fedor Moiseev, Rico Sennrich, and Ivan Titov. 2019. "Analyzing Multi-Head Self-Attention: Specialized Heads Do the Heavy Lifting, the Rest Can Be Pruned." *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics*. https://arxiv.org/abs/1905.09418.

Volkov, Vasily. 2016. "Understanding Latency Hiding on GPUs." PhD thesis, University of California, Berkeley.

Vries, Harm de, Florian Strub, Jérémie Mary, Hugo Larochelle, Olivier Pietquin, and Aaron Courville. 2017. "Modulating Early Visual Processing by Language." *Advances in Neural Information Processing Systems* 30. https://arxiv.org/abs/1707.00683.

Vyas, Nikhil, Depen Morwani, Rosie Zhao, et al. 2024. "SOAP: Improving and Stabilizing Shampoo Using Adam." *arXiv Preprint arXiv:2409.11321*.

Waleffe, Roger, Wonmin Byeon, Duncan Riach, et al. 2024. "An Empirical Study of Mamba-Based Language Models." *arXiv Preprint arXiv:2406.07887*.

Wallace, Bram, Meihua Dang, Rafael Rafailov, et al. 2024. "Diffusion Model Alignment Using Direct Preference Optimization." *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 8228–38. https://arxiv.org/abs/2311.12908.

Wallace, Bram, Akash Gokul, and Nikhil Naik. 2023. "EDICT: Exact Diffusion Inversion via Coupled Transformations." *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*.

Wang, Guanghan, Yair Schiff, Subham Sekhar Sahoo, and Volodymyr Kuleshov. 2025. "Remasking Discrete Diffusion Models with Inference-Time Scaling." *Advances in Neural Information Processing Systems*.

Wang, Hongyu, Shuming Ma, Li Dong, Shaohan Huang, Dongdong Zhang, and Furu Wei. 2024. "DeepNet: Scaling Transformers to 1,000 Layers." *IEEE Transactions on Pattern Analysis and Machine Intelligence* 46: 6761–74. https://arxiv.org/abs/2203.00555.

Wang, Jie, Tao Ji, Yuanbin Wu, et al. 2024. "Length Generalization of Causal Transformers Without Position Encoding." *Findings of the Association for Computational Linguistics: ACL 2024*, 14024–40. https://doi.org/10.18653/v1/2024.findings-acl.834.

Wang, Junxiong, Daniele Paliotta, Avner May, Alexander M. Rush, and Tri Dao. 2024. "The Mamba in the Llama: Distilling and Accelerating Hybrid Models." *Advances in Neural Information Processing Systems*.

Wang, Ke Alexander, Jiaxin Shi, and Emily B. Fox. 2025. "Test-Time Regression: A Unifying Framework for Designing Sequence Models with Associative Memory." *arXiv Preprint arXiv:2501.12352*.

Wang, Kevin, Alexandre Variengien, Arthur Conmy, Buck Shlegeris, and Jacob Steinhardt. 2023. "Interpretability in the Wild: A Circuit for Indirect Object Identification in GPT-2 Small." *International Conference on Learning Representations*. https://arxiv.org/abs/2211.00593.

Wang, Lean, Huazuo Gao, Chenggang Zhao, Xu Sun, and Damai Dai. 2024. "Auxiliary-Loss-Free Load Balancing Strategy for Mixture-of-Experts." *arXiv Preprint arXiv:2408.15664*.

Wang, Qiang, Bei Li, Tong Xiao, et al. 2019. "Learning Deep Transformer Models for Machine Translation." *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics*, 1810–22. https://doi.org/10.18653/v1/p19-1176.

Wang, Shuche, Fengzhuo Zhang, Jiaxiang Li, et al. 2025. "Muon Outperforms Adam in Tail-End Associative Memory Learning." *arXiv Preprint arXiv:2509.26030*.

Wang, Sinong, Belinda Z. Li, Madian Khabsa, Han Fang, and Hao Ma. 2020. "Linformer: Self-Attention with Linear Complexity." *arXiv Preprint arXiv:2006.04768*.

Wang, Thomas, Adam Roberts, Daniel Hesslow, et al. 2022. "What Language Model Architecture and Pretraining Objective Work Best for Zero-Shot Generalization?" *International Conference on Machine Learning*. https://arxiv.org/abs/2204.05832.

Wang, Xintao, Ke Yu, Shixiang Wu, et al. 2018. "ESRGAN: Enhanced Super-Resolution Generative Adversarial Networks." *European Conference on Computer Vision Workshops*. https://arxiv.org/abs/1809.00219.

Wang, Ziyu, Tom Schaul, Matteo Hessel, Hado van Hasselt, Marc Lanctot, and Nando de Freitas. 2016. "Dueling Network Architectures for Deep Reinforcement Learning." *International Conference on Machine Learning*, 1995–2003.

Warner, Benjamin, Antoine Chaffin, Benjamin Clavié, et al. 2024. "Smarter, Better, Faster, Longer: A Modern Bidirectional Encoder for Fast, Memory Efficient, and Long Context Finetuning and Inference." *arXiv Preprint arXiv:2412.13663*.

Watkins, Christopher J C H. 1989. "Learning from Delayed Rewards." PhD thesis, King's College, University of Cambridge. http://www.cs.rhul.ac.uk/~chrisw/new_thesis.pdf.

Watkins, Christopher JCH, and Peter Dayan. 1992. "Q-Learning." *Machine Learning* 8 (3–4):

279–92. https://doi.org/10.1007/bf00992698.

Watson, Geoffrey S. 1964. "Smooth Regression Analysis." *Sankhyā: The Indian Journal of Statistics, Series A*, 359–72. https://doi.org/10.1007/bf02868765.

Weed, Jonathan, and Francis Bach. 2019. "Sharp Asymptotic and Finite-Sample Rates of Convergence of Empirical Measures in Wasserstein Distance." *Bernoulli* 25 (4A): 2620–48.

Wei, Jason, Yi Tay, Rishi Bommasani, et al. 2022. "Emergent Abilities of Large Language Models." *ArXiv:2206.07682*. https://arxiv.org/abs/2206.07682.

Weinberger, Kilian Q., and Gerald Tesauro. 2007. "Metric Learning for Kernel Regression." *Proceedings of the 11th International Conference on Artificial Intelligence and Statistics*, Proceedings of machine learning research, vol. 2: 612–19.

Welch, Lloyd R. 1974. "Lower Bounds on the Maximum Cross Correlation of Signals." *IEEE Transactions on Information Theory* 20 (3): 397–99. https://doi.org/10.1109/TIT.1974.1055219.

Welling, Max, and Yee W Teh. 2011. "Bayesian Learning via Stochastic Gradient Langevin Dynamics." *Proceedings of the 28th International Conference on Machine Learning (ICML-11)*, 681–88. https://dl.acm.org/doi/10.5555/3104482.3104568.

Wen, Kaiyue, David Hall, Tengyu Ma, and Percy Liang. 2025. "Fantastic Pretraining Optimizers and Where to Find Them." *arXiv Preprint arXiv:2509.02046*.

Wen, Kaiyue, Zhiyuan Li, Jason Wang, David Hall, Percy Liang, and Tengyu Ma. 2024. "Understanding Warmup-Stable-Decay Learning Rates: A River Valley Loss Landscape Perspective." *arXiv Preprint arXiv:2410.05192*.

Weng, Lilian. 2021. *What Are Diffusion Models?* Blog post, https://lilianweng.github.io/posts/2021-07-11-diffusion-models/.

Wenliang, Li K., and Heishiro Kanagawa. 2020. "Blindness of Score-Based Methods to Isolated Components and Mixing Proportions." *arXiv Preprint arXiv:2008.10087*. https://arxiv.org/abs/2008.10087.

Wettig, Alexander, Tianyu Gao, Zexuan Zhong, and Danqi Chen. 2023. "Should You Mask 15% in Masked Language Modeling?" *Proceedings of the 17th Conference of the European Chapter of the Association for Computational Linguistics*, 2985–3000.

Widrow, Bernard, and Marcian E. Hoff. 1960. "Adaptive Switching Circuits." *IRE WESCON Convention Record* 4: 96–104.

Wiegreffe, Sarah, and Yuval Pinter. 2019. "Attention Is Not Not Explanation." *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing*, 11–20.

Wigner, Eugene P. 1958. "On the Distribution of the Roots of Certain Symmetric Matrices." *Ann. Math.*, 325–27. https://doi.org/10.2307/1970079.

Williams, Ronald J. 1992. "Simple Statistical Gradient-Following Algorithms for Connectionist Reinforcement Learning." *Machine Learning* 8: 229–56.

Williams, Samuel, Andrew Waterman, and David Patterson. 2009. *Roofline: An Insightful Visual Performance Model for Floating-Point Programs and Multicore Architectures*. Lawrence Berkeley National Lab. https://doi.org/10.2172/1407078.

Wortsman, Mitchell, Gabriel Ilharco, Samir Yitzhak Gadre, et al. 2022. "Model Soups: Averaging Weights of Multiple Fine-Tuned Models Improves Accuracy Without Increasing Inference Time." *International Conference on Machine Learning*.

Wortsman, Mitchell, Peter J. Liu, Lechao Xiao, et al. 2024. "Small-Scale Proxies for Large-Scale Transformer Training Instabilities." *International Conference on Learning Representations*. https://arxiv.org/abs/2309.14322.

Wu, Chengyue, Hao Zhang, Shuchen Xue, et al. 2026. "Fast-dLLM: Training-Free Acceleration of Diffusion LLM by Enabling KV Cache and Parallel Decoding." *International Conference on Learning Representations*.

Xiao, Guangxuan, Yuandong Tian, Beidi Chen, Song Han, and Mike Lewis. 2024. "Efficient Streaming Language Models with Attention Sinks." *International Conference on Learning Representations*.

Xiong, Ruibin, Yunchang Yang, Di He, et al. 2020. "On Layer Normalization in the Transformer Architecture." *International Conference on Machine Learning*, 10524–33. https://proceedings.mlr.press/v119/xiong20b.html.

Xiong, Wenhan, Jingyu Liu, Igor Molybog, et al. 2024. "Effective Long-Context Scaling of Foundation Models." *Proceedings of the 2024 Conference of the North American Chapter of the Association for Computational Linguistics*, 4643–63. https://doi.org/10.18653/v1/2024.naacl-long.260.

Xu, Yilun, Mingyang Deng, Xiang Cheng, Yonglong Tian, Ziming Liu, and Tommi Jaakkola. 2023. "Restart Sampling for Improving Generative Processes." *Advances in Neural Information Processing Systems* 36.

Xu, Yuanzhong, HyoukJoong Lee, Dehao Chen, et al. 2021. "GSPMD: General and Scalable Parallelization for ML Computation Graphs." *ArXiv:2105.04663*. https://arxiv.org/abs/2105.04663.

Yang, An, Anfeng Li, Baosong Yang, et al. 2025. "Qwen3 Technical Report." *arXiv Preprint arXiv:2505.09388*.

Yang, Charlene, Yunsong Wang, Steven Farrell, Thorsten Kurth, and Samuel Williams. 2020. "Hierarchical Roofline Performance Analysis for Deep Learning Applications." *arXiv Preprint arXiv:2009.05257*.

Yang, Greg, and Edward J. Hu. 2021. "Tensor Programs IV: Feature Learning in Infinite-Width Neural Networks." *International Conference on Machine Learning*, 11727–37.

Yang, Greg, Edward J. Hu, Igor Babuschkin, et al. 2022. "Tensor Programs V: Tuning Large Neural Networks via Zero-Shot Hyperparameter Transfer." *arXiv Preprint arXiv:2203.03466*.

Yang, Greg, James B. Simon, and Jeremy Bernstein. 2023. "A Spectral Condition for Feature Learning." *arXiv Preprint arXiv:2310.17813*.

Yang, Songlin, Jan Kautz, and Ali Hatamizadeh. 2025. "Gated Delta Networks: Improving Mamba2 with Delta Rule." *International Conference on Learning Representations*. https://arxiv.org/abs/2412.06464.

Yang, Songlin, Bailin Wang, Yikang Shen, Rameswar Panda, and Yoon Kim. 2024. "Gated Linear Attention Transformers with Hardware-Efficient Training." *International Conference on Machine Learning*, 56501–23. https://arxiv.org/abs/2312.06635.

Yang, Songlin, Bailin Wang, Yu Zhang, Yikang Shen, and Yoon Kim. 2024. "Parallelizing Linear Transformers with the Delta Rule over Sequence Length." *Advances in Neural Information Processing Systems* 37. https://arxiv.org/abs/2406.06484.

Ye, Jiacheng, Zhihui Xie, Lin Zheng, et al. 2025. "Dream 7B: Diffusion Large Language Models." *arXiv Preprint arXiv:2508.15487*. https://arxiv.org/abs/2508.15487.

Ye, Tianzhu, Li Dong, Yuqing Xia, et al. 2025. "Differential Transformer." *International Conference on Learning Representations*. https://arxiv.org/abs/2410.05258.

Yin, Tianwei, Michaël Gharbi, Taesung Park, et al. 2024. "Improved Distribution Matching Distillation for Fast Image Synthesis." *Advances in Neural Information Processing Systems* 37.

Yin, Tianwei, Michaël Gharbi, Richard Zhang, et al. 2024. "One-Step Diffusion with Distribution Matching Distillation." *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 6613–23. https://arxiv.org/abs/2311.18828.

Yu, Gyeong-In, Joo Seong Jeong, Geon-Woo Kim, Soojeong Kim, and Byung-Gon Chun. 2022. "Orca: A Distributed Serving System for Transformer-Based Generative Models." *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, 521–38. https://www.usenix.org/conference/osdi22/presentation/yu.

Yu, Qiying, Zheng Zhang, Ruofei Zhu, et al. 2025. "DAPO: An Open-Source LLM Reinforcement Learning System at Scale." *arXiv Preprint arXiv:2503.14476*.

Yu, Sihyun, Sangkyung Kwak, Huiwon Jang, et al. 2025. "Representation Alignment for Generation: Training Diffusion Transformers Is Easier Than You Think." *International Conference on Learning Representations*. https://arxiv.org/abs/2410.06940.

Yu, Tianhe, Garrett Thomas, Lantao Yu, et al. 2020. "MOPO: Model-Based Offline Policy Optimization." *Advances in Neural Information Processing Systems* 33.

Yuan, Jingyang, Huazuo Gao, Damai Dai, et al. 2025. "Native Sparse Attention: Hardware-Aligned and Natively Trainable Sparse Attention." *Proceedings of the 63rd Annual Meeting*

of the Association for Computational Linguistics (Volume 1: Long Papers), 23078–97. https://doi.org/10.18653/v1/2025.acl-long.1126.

Zaheer, Manzil, Sashank Reddi, Devendra Sachan, Satyen Kale, and Sanjiv Kumar. 2018. "Adaptive Methods for Nonconvex Optimization." *Advances in Neural Information Processing Systems*, 9793–803. https://proceedings.neurips.cc/paper/2018/hash/90365351ccc7437a1309dc64e4db32a3-Abstract.html.

Zeiler, Matthew D. 2012. "ADADELTA: An Adaptive Learning Rate Method." *ArXiv:1212.5701*. https://arxiv.org/abs/1212.5701.

Zeng, Aohan, Xin Lv, Qinkai Zheng, Zhenyu Hou, Bin Chen, et al. 2025. "GLM-4.5: Agentic, Reasoning, and Coding (ARC) Foundation Models." *arXiv Preprint arXiv:2508.06471*.

Zhai, Xiaohua, Basil Mustafa, Alexander Kolesnikov, and Lucas Beyer. 2023. "Sigmoid Loss for Language Image Pre-Training." *Proceedings of the IEEE/CVF International Conference on Computer Vision*, 11941–52. https://arxiv.org/abs/2303.15343.

Zhang, Biao, Fedor Moiseev, Joshua Ainslie, et al. 2025. "Encoder-Decoder Gemma: Improving the Quality-Efficiency Trade-Off via Adaptation." *arXiv Preprint arXiv:2504.06225*.

Zhang, Biao, and Rico Sennrich. 2019. "Root Mean Square Layer Normalization." *Advances in Neural Information Processing Systems* 32.

Zhang, Fred, and Neel Nanda. 2024. "Towards Best Practices of Activation Patching in Language Models: Metrics and Methods." *International Conference on Learning Representations*. https://arxiv.org/abs/2309.16042.

Zhang, Han, Ian Goodfellow, Dimitris Metaxas, and Augustus Odena. 2019. "Self-Attention Generative Adversarial Networks." *Proceedings of the 36th International Conference on Machine Learning*, 7354–63. https://arxiv.org/abs/1805.08318.

Zhang, Hanlin, Depen Morwani, Nikhil Vyas, et al. 2024. "How Does Critical Batch Size Scale in Pre-Training?" *arXiv Preprint arXiv:2410.21676*.

Zhang, Jack, Noah Amsel, Berlin Chen, and Tri Dao. 2026. *Gram Newton–Schulz: A Fast, Hardware-Aware Newton–Schulz Algorithm for Muon*. Https://tridao.me/blog/2026/gram-newton-schulz/.

Zhang, Jingzhao, Sai Praneeth Karimireddy, Andreas Veit, et al. 2020. "Why Are Adaptive Methods Good for Attention Models?" *Advances in Neural Information Processing Systems* 33.

Zhang, Lvmin, Anyi Rao, and Maneesh Agrawala. 2023. "Adding Conditional Control to Text-to-Image Diffusion Models." *Proceedings of the IEEE/CVF International Conference on Computer Vision*, 3836–47. https://arxiv.org/abs/2302.05543.

Zhang, Susan, Stephen Roller, Naman Goyal, et al. 2022. "OPT: Open Pre-Trained Transformer Language Models." *ArXiv:2205.01068*. https://arxiv.org/abs/2205.01068.

Zhang, Yushun, Congliang Chen, Tian Ding, Ziniu Li, Ruoyu Sun, and Zhi-Quan Luo. 2024. "Why Transformers Need Adam: A Hessian Perspective." *Advances in Neural Information Processing Systems* 37. https://arxiv.org/abs/2402.16788.

Zhang, Zhenyu, Ying Sheng, Tianyi Zhou, et al. 2023. "H_2O: Heavy-Hitter Oracle for Efficient Generative Inference of Large Language Models." *Advances in Neural Information Processing Systems* 36. https://arxiv.org/abs/2306.14048.

Zhao, Shengyu, Zhijian Liu, Ji Lin, Jun-Yan Zhu, and Song Han. 2020. "Differentiable Augmentation for Data-Efficient GAN Training." *Advances in Neural Information Processing Systems* 33.

Zhao, Wenliang, Lujia Bai, Yongming Rao, Jie Zhou, and Jiwen Lu. 2023. "UniPC: A Unified Predictor-Corrector Framework for Fast Sampling of Diffusion Models." *Advances in Neural Information Processing Systems* 36. https://arxiv.org/abs/2302.04867.

Zhao, Yanli, Andrew Gu, Rohan Varma, et al. 2023. "PyTorch FSDP: Experiences on Scaling Fully Sharded Data Parallel." *Proceedings of the VLDB Endowment* 16 (12): 3848–60. https://doi.org/10.14778/3611540.3611569.

Zheng, Boyang, Nanye Ma, Shengbang Tong, and Saining Xie. 2026. "Diffusion Transformers with Representation Autoencoders." *International Conference on Learning Representations*. https://arxiv.org/abs/2510.11690.

Zheng, Chujie, Shixuan Liu, Mingze Li, et al. 2025. "Group Sequence Policy Optimization." *arXiv Preprint arXiv:2507.18071*.

Zheng, Kaiwen, Yongxin Chen, Huayu Chen, et al. 2025. "Direct Discriminative Optimization: Your Likelihood-Based Visual Generative Model Is Secretly a GAN Discriminator." *Proceedings of the 42nd International Conference on Machine Learning*.

Zheng, Kaiwen, Yongxin Chen, Hanzi Mao, Ming-Yu Liu, Jun Zhu, and Qinsheng Zhang. 2025. "Masked Diffusion Models Are Secretly Time-Agnostic Masked Models and Exploit Inaccurate Categorical Sampling." *International Conference on Learning Representations*.

Zheng, Kaiwen, Cheng Lu, Jianfei Chen, and Jun Zhu. 2023. "Improved Techniques for Maximum Likelihood Estimation for Diffusion ODEs." *International Conference on Machine Learning*. https://arxiv.org/abs/2305.03935.

Zheng, Lianmin, Liangsheng Yin, Zhiqiang Xie, et al. 2024. "SGLang: Efficient Execution of Structured Language Model Programs." *Advances in Neural Information Processing Systems* 37. https://arxiv.org/abs/2312.07104.

Zhou, Yanqi, Tao Lei, Hanxiao Liu, et al. 2022. "Mixture-of-Experts with Expert Choice Routing." *Advances in Neural Information Processing Systems* 35: 7103–14.

Zhu, Jiachen, Xinlei Chen, Kaiming He, Yann LeCun, and Zhuang Liu. 2025. "Transformers Without Normalization." *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 14901–11. https://arxiv.org/abs/2503.10622.

Zhu, Jun-Yan, Taesung Park, Phillip Isola, and Alexei A Efros. 2017. "Unpaired Image-to-Image Translation Using Cycle-Consistent Adversarial Networks." *Proceedings of the IEEE International Conference on Computer Vision*, 2223–32. https://doi.org/10.1109/iccv.2017.244.

Zhu, Lianghui, Bencheng Liao, Qian Zhang, Xinlong Wang, Wenyu Liu, and Xinggang Wang. 2024. "Vision Mamba: Efficient Visual Representation Learning with Bidirectional State Space Model." *International Conference on Machine Learning*. https://arxiv.org/abs/2401.09417.

Ziebart, Brian D, Andrew Maas, J Andrew Bagnell, and Anind K Dey. 2008. "Maximum Entropy Inverse Reinforcement Learning." *Proceedings of the AAAI Conference on Artificial Intelligence*, 1433–38.

Zoph, Barret, Irwan Bello, Sameer Kumar, et al. 2022. "ST-MoE: Designing Stable and Transferable Sparse Expert Models." *arXiv Preprint arXiv:2202.08906*.