

Dive into Deep Learning

BASICS

Alexander J Smola

Aston Zhang · Zachary C Lipton · Mu Li · Alexander J Smola (1st edition)

Table of contents

Basics **1**

1 Preliminaries **3**
Resources and Further Reading . . . 4
1.1 Data Manipulation . . . 4
1.1.1 Tensor Creation . . . 5
1.1.2 Reshaping and Indexing . . . 8
1.1.3 Operations . . . 11
1.1.4 Memory and Conversion . . . 15
1.1.5 Summary . . . 17
1.1.6 Exercises . . . 17
1.2 Data Preprocessing . . . 18
1.2.1 Loading and Inspecting . . . 19
1.2.2 Cleaning and Encoding . . . 21
1.2.3 Scaling and Converting . . . 23
1.2.4 Summary . . . 25
1.2.5 Exercises . . . 25
1.3 Linear Algebra . . . 26
1.3.1 Scalars, Vectors, Matrices, and Tensors . . . 28
1.3.2 Elementwise Operations and Reductions . . . 30
1.3.3 Products . . . 32
1.3.4 Norms . . . 35
1.3.5 Eigenvectors and Eigenvalues . . . 39
1.3.6 Summary . . . 42
1.3.7 Exercises . . . 42
1.4 Calculus . . . 45
1.4.1 Derivatives . . . 45
1.4.2 Partial Derivatives and Gradients . . . 50
1.4.3 The Chain Rule . . . 56
1.4.4 Summary . . . 58
1.4.5 Exercises . . . 58
1.5 Automatic Differentiation . . . 60
1.5.1 Computing Gradients . . . 61
1.5.2 Controlling the Graph . . . 63
1.5.3 Differentiating Programs . . . 65
1.5.4 Forward and Reverse Mode . . . 66
1.5.5 Summary . . . 68

1.5.6 Exercises . . . 68
1.6 Probability and Statistics . . . 69
1.6.1 Sampling . . . 70
1.6.2 Events and Random Variables . . . 72
1.6.3 Expectation and Variance . . . 75
1.6.4 Estimation Error . . . 77
1.6.5 Conditional Probability . . . 80
1.6.6 Summary . . . 85
1.6.7 Exercises . . . 85
1.7 Documentation . . . 87
1.7.1 Discovering Names . . . 88
1.7.2 Reading Documentation . . . 88
1.7.3 Verifying Behavior . . . 89
1.7.4 Summary . . . 90
1.7.5 Exercises . . . 90

2 Linear Regression **93**
Resources and Further Reading . . . 94
2.1 Linear Regression . . . 95
2.1.1 The Model and the Loss . . . 95
2.1.2 Minimizing the Loss . . . 98
2.1.3 Squared Loss and Gaussian Noise . . . 103
2.1.4 Linear Regression as a Neural Network . . . 106
2.1.5 Summary . . . 107
2.1.6 Exercises . . . 107
2.2 Synthetic Data . . . 110
2.2.1 Generating the Data . . . 110
2.2.2 Recovering the Parameters . . . 111
2.2.3 Summary . . . 112
2.2.4 Exercises . . . 113
2.3 Implementation from Scratch . . . 114
2.3.1 Model, Loss, and Gradient . . . 114
2.3.2 Training . . . 116
2.3.3 Checking the Implementation . . . 118
2.3.4 Summary . . . 121
2.3.5 Exercises . . . 121
2.4 Object-Oriented Design for Implementation . . . 123
2.4.1 Utilities . . . 124
2.4.2 Models . . . 125
2.4.3 Data . . . 127
2.4.4 Training . . . 129
2.4.5 Training Linear Regression Again . . . 133
2.4.6 Summary . . . 134
2.4.7 Exercises . . . 134
2.5 Concise Implementation . . . 135
2.5.1 The Fully Connected Layer . . . 136
2.5.2 The Loss Function . . . 136
2.5.3 The Optimizer . . . 137
2.5.4 Training . . . 137
2.5.5 Summary . . . 139

2.5.6 Exercises . . . 139
2.6 Generalization . . . 140
2.6.1 Training Error and Generalization Error . . . 141
2.6.2 Underfitting and Overfitting . . . 142
2.6.3 Model Selection . . . 149
2.6.4 Summary . . . 152
2.6.5 Exercises . . . 152
2.7 Weight Decay . . . 154
2.7.1 Norm Penalties . . . 154
2.7.2 Weight Decay from Scratch . . . 157
2.7.3 Shrinkage Along Eigenvectors . . . 160
2.7.4 Concise Implementation . . . 165
2.7.5 Summary . . . 166
2.7.6 Exercises . . . 166

3 Linear Classification **171**
Resources and Further Reading . . . 172
3.1 Softmax Regression . . . 172
3.1.1 The Model . . . 173
3.1.2 The Cross-Entropy Loss . . . 176
3.1.3 Accuracy and Confidence . . . 180
3.1.4 Summary . . . 182
3.1.5 Exercises . . . 182
3.2 Image Classification Data . . . 185
3.2.1 Loading the Dataset . . . 185
3.2.2 Reading Minibatches . . . 186
3.2.3 Inspecting the Images . . . 187
3.2.4 Summary . . . 188
3.2.5 Exercises . . . 189
3.3 Softmax Regression from Scratch . . . 190
3.3.1 The Softmax . . . 190
3.3.2 Loss and Accuracy . . . 191
3.3.3 The Model and the Loss . . . 193
3.3.4 Training and Evaluation . . . 195
3.3.5 Summary . . . 200
3.3.6 Exercises . . . 201
3.4 Concise Implementation . . . 202
3.4.1 The Model . . . 202
3.4.2 A Stable Cross-Entropy Loss . . . 203
3.4.3 Training . . . 204
3.4.4 Summary . . . 205
3.4.5 Exercises . . . 205
3.5 Measuring Classifiers . . . 206
3.5.1 The Limits of Accuracy . . . 207
3.5.2 Calibration . . . 208
3.5.3 Summary . . . 212
3.5.4 Exercises . . . 212
3.6 Generalization in Classification . . . 215
3.6.1 The Test Set . . . 215
3.6.2 Test Set Reuse . . . 218

3.6.3 Statistical Learning Theory . . . 220
3.6.4 Summary . . . 223
3.6.5 Exercises . . . 223
3.7 Environment and Distribution Shift . . . 225
3.7.1 Types of Distribution Shift . . . 225
3.7.2 Covariate Shift Correction . . . 228
3.7.3 Label Shift Correction . . . 235
3.7.4 Concept Shift and Feedback . . . 238
3.7.5 Summary . . . 239
3.7.6 Exercises . . . 239

4 Multilayer Perceptron 243
Resources and Further Reading . . . 244
4.1 Multilayer Perceptrons . . . 245
4.1.1 Limits of Linear Models . . . 245
4.1.2 Hidden Layers . . . 246
4.1.3 Expressive Power . . . 249
4.1.4 Activation Functions . . . 251
4.1.5 Summary . . . 253
4.1.6 Exercises . . . 254
4.2 Implementation . . . 255
4.2.1 Implementation from Scratch . . . 256
4.2.2 Concise Implementation . . . 260
4.2.3 Comparison with Softmax Regression . . . 260
4.2.4 Summary . . . 264
4.2.5 Exercises . . . 265
4.3 Backpropagation . . . 266
4.3.1 Computational Graphs . . . 267
4.3.2 Backpropagation through an MLP . . . 268
4.3.3 Memory and Computation . . . 273
4.3.4 Summary . . . 274
4.3.5 Exercises . . . 274
4.4 Numerical Stability and Initialization . . . 275
4.4.1 Vanishing and Exploding Gradients . . . 276
4.4.2 Parameter Initialization . . . 279
4.4.3 Symmetry among Hidden Units . . . 283
4.4.4 Summary . . . 284
4.4.5 Exercises . . . 285
4.5 Generalization in Deep Learning . . . 287
4.5.1 Memorization and Capacity . . . 287
4.5.2 Double Descent . . . 290
4.5.3 Implicit Bias of Gradient Descent . . . 293
4.5.4 Early Stopping . . . 296
4.5.5 Summary . . . 298
4.5.6 Exercises . . . 298
4.6 Dropout . . . 300
4.6.1 Noise in Hidden Layers . . . 301
4.6.2 Implementation . . . 302
4.6.3 Effects of Dropout . . . 305
4.6.4 Interpretations . . . 307

4.6.5 Summary . . . 308
4.6.6 Exercises . . . 308
4.7 Predicting House Prices on Kaggle . . . 310
4.7.1 The Competition . . . 310
4.7.2 Preprocessing . . . 312
4.7.3 Model Selection . . . 314
4.7.4 Submitting Predictions . . . 317
4.7.5 Summary . . . 318
4.7.6 Exercises . . . 319

5 Computation **323**
Resources and Further Reading . . . 324
5.1 Modules and Model Construction . . . 325
5.1.1 The Module Abstraction . . . 325
5.1.2 Module Containers . . . 328
5.1.3 Forward Computation . . . 330
5.1.4 Lazy Initialization . . . 332
5.1.5 Models from a Config . . . 333
5.1.6 Summary . . . 335
5.1.7 Exercises . . . 335
5.2 Parameters, State, and Memory . . . 336
5.2.1 The State Tree . . . 337
5.2.2 Training Memory . . . 339
5.2.3 Shared, Frozen, and Averaged Parameters . . . 341
5.2.4 Summary . . . 345
5.2.5 Exercises . . . 346
5.3 Initialization . . . 347
5.3.1 Library Defaults . . . 347
5.3.2 Applying Initializers . . . 348
5.3.3 Custom Initializers and Direct Edits . . . 349
5.3.4 Initialization of Residual Networks . . . 350
5.3.5 Summary . . . 353
5.3.6 Exercises . . . 354
5.4 Custom Layers and Functions . . . 355
5.4.1 Layers without Parameters . . . 356
5.4.2 Layers with Parameters . . . 356
5.4.3 Layers with Fixed State . . . 359
5.4.4 Functions with Custom Gradients . . . 361
5.4.5 Summary . . . 364
5.4.6 Exercises . . . 364
5.5 Numerical Formats . . . 365
5.5.1 Floating-Point Formats . . . 366
5.5.2 Dtype Promotion and Model Casting . . . 368
5.5.3 Mixed-Precision Training . . . 369
5.5.4 Numerical Failure Modes . . . 373
5.5.5 Summary . . . 375
5.5.6 Exercises . . . 375
5.6 Saving, Loading, and Pretrained Weights . . . 376
5.6.1 Model State . . . 377
5.6.2 Pickle and safetensors . . . 378

5.6.3 Training Checkpoints . . . 380
5.6.4 Pretrained Weights . . . 384
5.6.5 Summary . . . 386
5.6.6 Exercises . . . 386
5.7 GPUs, Devices, and Memory . . . 387
5.7.1 Devices and Placement . . . 388
5.7.2 Asynchronous Execution . . . 390
5.7.3 GPU Memory . . . 393
5.7.4 Device-Aware Trainer . . . 398
5.7.5 Summary . . . 399
5.7.6 Exercises . . . 400
5.8 Reproducibility and Inspection . . . 401
5.8.1 Seeds and Random Number Generators . . . 402
5.8.2 Deterministic Arithmetic . . . 405
5.8.3 Model Inspection with Hooks . . . 406
5.8.4 Summary . . . 410
5.8.5 Exercises . . . 410

6 Convolutional Neural Networks 413
Resources and Further Reading . . . 414
6.1 From Fully Connected Layers to Convolutions . . . 415
6.1.1 Equivariance and Invariance . . . 416
6.1.2 Constraints on the Fully Connected Layer . . . 418
6.1.3 Convolution and Cross-Correlation . . . 420
6.1.4 Channels . . . 421
6.1.5 Summary . . . 422
6.1.6 Exercises . . . 422
6.2 Convolutions for Images . . . 424
6.2.1 Cross-Correlation . . . 424
6.2.2 Convolutional Layers . . . 426
6.2.3 Convolution as Matrix Multiplication . . . 429
6.2.4 Feature Maps and Receptive Fields . . . 429
6.2.5 Summary . . . 431
6.2.6 Exercises . . . 431
6.3 Padding, Stride, and Dilation . . . 432
6.3.1 Padding . . . 433
6.3.2 Stride . . . 435
6.3.3 Dilation . . . 437
6.3.4 Summary . . . 439
6.3.5 Exercises . . . 439
6.4 Multiple Input and Multiple Output Channels . . . 441
6.4.1 Multiple Input Channels . . . 441
6.4.2 Multiple Output Channels . . . 442
6.4.3 The 1×1 Convolution . . . 444
6.4.4 Summary . . . 445
6.4.5 Exercises . . . 446
6.5 Pooling . . . 446
6.5.1 Maximum Pooling and Average Pooling . . . 447
6.5.2 Tolerance to Small Shifts . . . 448
6.5.3 Global Pooling . . . 450

6.5.4 Padding and Stride . . . 450
6.5.5 Multiple Channels . . . 452
6.5.6 Summary . . . 452
6.5.7 Exercises . . . 452
6.6 LeNet . . . 454
6.6.1 The LeNet Architecture . . . 454
6.6.2 Layer Shapes . . . 456
6.6.3 Parameters and Multiply-Adds . . . 456
6.6.4 Training . . . 458
6.6.5 Summary . . . 462
6.6.6 Exercises . . . 462

7 Modern Convnets 465
Resources and Further Reading . . . 466
7.1 AlexNet and Learned Image Representations . . . 467
7.1.1 Learned Features, Data, and Hardware . . . 467
7.1.2 The AlexNet Architecture . . . 469
7.1.3 Training on Fashion-MNIST . . . 473
7.1.4 Summary . . . 476
7.1.5 Exercises . . . 476
7.2 Blocks, Bottlenecks, and Branches: VGG, NiN, GoogLeNet . . . 477
7.2.1 VGG: Blocks as the Unit of Design . . . 478
7.2.2 NiN: 1×1 Convolutions and Global Average Pooling . . . 482
7.2.3 GoogLeNet: Multi-Branch Blocks . . . 486
7.2.4 Comparing the Networks . . . 489
7.2.5 Summary . . . 491
7.2.6 Exercises . . . 491
7.3 Normalization Layers . . . 492
7.3.1 Batch Normalization . . . 493
7.3.2 Implementation from Scratch . . . 495
7.3.3 LeNet with Batch Normalization . . . 497
7.3.4 Normalization without Batch Statistics . . . 500
7.3.5 Summary . . . 502
7.3.6 Exercises . . . 502
7.4 Residual Networks . . . 504
7.4.1 Function Classes and Depth . . . 504
7.4.2 Residual Blocks . . . 505
7.4.3 Plain and Residual Networks Compared . . . 508
7.4.4 ResNet-18 . . . 511
7.4.5 ResNeXt and DenseNet . . . 514
7.4.6 Summary . . . 518
7.4.7 Exercises . . . 518
7.5 Efficient Convnets . . . 521
7.5.1 Depthwise-Separable Networks . . . 521
7.5.2 Scaling and Searching . . . 532
7.5.3 Structural Re-parameterization . . . 533
7.5.4 Summary . . . 536
7.5.5 Exercises . . . 536
7.6 Modern Training Recipes for Convnets . . . 538
7.6.1 Training Recipes from 2015 to 2022 . . . 539

7.6.2 Implementing the Ingredients . . . 540
7.6.3 Two Recipes on One Network . . . 545
7.6.4 Comparing Published Results . . . 549
7.6.5 Summary . . . 550
7.6.6 Exercises . . . 550
7.7 ConvNeXt: Modernizing a ResNet Step by Step . . . 551
7.7.1 The Modernization Roadmap . . . 552
7.7.2 Implementation . . . 554
7.7.3 A Small Roadmap on Fashion-MNIST . . . 558
7.7.4 Limits and Open Problems . . . 561
7.7.5 Summary . . . 562
7.7.6 Exercises . . . 562
7.8 Convolutional Network Design Spaces . . . 563
7.8.1 The AnyNet Design Space . . . 564
7.8.2 Comparing Design Spaces . . . 568
7.8.3 RegNet . . . 572
7.8.4 Summary . . . 575
7.8.5 Exercises . . . 576

8 Sequence Models **579**
Resources and Further Reading . . . 580
8.1 Working with Sequences . . . 581
8.1.1 Sequential Data . . . 582
8.1.2 Autoregressive Models . . . 582
8.1.3 Markov Models and Stationarity . . . 584
8.1.4 Forecasting a Noisy Sine Wave . . . 585
8.1.5 Summary . . . 590
8.1.6 Exercises . . . 591
8.2 From Text to Tokens . . . 592
8.2.1 Reading the Dataset . . . 592
8.2.2 Characters, Words, and Bytes . . . 593
8.2.3 Byte Pair Encoding . . . 596
8.2.4 Special Tokens . . . 603
8.2.5 Tokenizers in the Wild . . . 604
8.2.6 Summary . . . 607
8.2.7 Exercises . . . 607
8.3 Language Models . . . 609
8.3.1 n-gram Language Models . . . 610
8.3.2 Perplexity and Smoothing . . . 615
8.3.3 Partitioning Sequences . . . 621
8.3.4 Summary . . . 623
8.3.5 Exercises . . . 623
8.4 Recurrent Neural Networks . . . 625
8.4.1 Neural Networks without Hidden States . . . 626
8.4.2 Recurrent Neural Networks with Hidden States . . . 626
8.4.3 RNN Language Models . . . 628
8.4.4 Summary . . . 629
8.4.5 Exercises . . . 631
8.5 Implementing RNNs . . . 632
8.5.1 An RNN Language Model from Scratch . . . 632

8.5.2 Gradient Clipping 636
8.5.3 Training 637
8.5.4 Text Generation 643
8.5.5 Concise Implementation 644
8.5.6 Summary 646
8.5.7 Exercises 647
8.6 Backpropagation Through Time 648
8.6.1 The Unrolled Computational Graph 649
8.6.2 Vanishing and Exploding Gradients 650
8.6.3 Truncated Backpropagation Through Time 655
8.6.4 Summary 656
8.6.5 Exercises 656
8.7 Decoding and Generation 658
8.7.1 The Decoding Problem 658
8.7.2 Greedy Decoding 660
8.7.3 Beam Search 661
8.7.4 Sampling 664
8.7.5 Evaluation and Cost 669
8.7.6 Summary 671
8.7.7 Exercises 671

References **673**

Basics

The introduction described programs that learn from examples instead of following rules written by hand. Suppose we want such a program for a small photograph of a garment: it should say whether the photograph shows a shirt, a sneaker, or a bag. Nobody knows how to write rules for this. We can, however, collect tens of thousands of labeled photographs. How do we turn those examples into a classifier we can trust on photographs it has never seen?

Our answer is a recipe with three ingredients: a model with parameters, a loss that measures the model's errors, and an algorithm that reduces the loss by changing the parameters. The recipe itself is two centuries old, and astronomers used it first (Chapter 2 tells the story). What has changed is the model. A deep network composes many simple functions, and automatic differentiation computes the derivatives of the loss with respect to all of their parameters, so that nobody has to derive them by hand. Nearly every model in this book, including the largest language models, is trained this way.

The first three chapters teach the recipe on models small enough to understand completely. Chapter 1 collects the tools: tensors, derivatives, and enough probability to reason about uncertainty. Chapter 2 fits a linear model to numbers and asks whether the fit will hold for new data, and Chapter 3 does the same for categories, which is where the garments return. The next two chapters make the model deep. Chapter 4 adds hidden layers and deals with the troubles they bring, and Chapter 5 turns to the software that assembles such a model, saves it, and runs it on a GPU. The last three chapters build what we know about the data into the model. Pixels have neighbors, which leads to convolutional networks (Chapter 6) and to the ideas that let them grow to more than a hundred layers (Chapter 7). Words come in order, which leads to models of text and other sequences (Chapter 8).

The Advanced part takes the engineering habits of this part (initialization, numerical stability, checkpointing, GPU memory) for granted. This part uses minibatch stochastic gradient descent, momentum, and learning-rate schedules as tools; Chapter 9 studies why they work.

Chapter 1

Preliminaries

The introduction described every learning problem by four components: data, a model that turns inputs into predictions, a loss that says how badly the model does, and an algorithm that adjusts the model's parameters to lower the loss. Now suppose that we had to write the program that sorts photographs of garments. What would it have to do, and what do we need to know to write it?

It would store the photographs as numbers and compute with all of them at once. It would combine those numbers into a score for each category, find out how the loss changes when each parameter changes, and judge whether its accuracy reflects a real pattern or the luck of the examples it saw.

Probability began in 1654, in letters between Pascal and Fermat about a game of chance, and calculus is more than three centuries old. A reader who feared that deep learning needs exotic mathematics may relax. Deep learning borrows old mathematics and applies it at a scale where calculation by hand is out of the question, and for that it needs two kinds of software. Array libraries apply one operation to millions of numbers in compiled code. Automatic differentiation computes the derivatives of a program. Every later chapter relies on both.

The chapter takes the program's needs in order. Two sections are about data: how to store it in tensors and compute with it (Section 1.1), and how to get there from a table of real records, with its missing entries and its text categories (Section 1.2). Three sections follow a prediction from a weighted sum to a gradient: linear algebra for the prediction and for the size of its error (Section 1.3), calculus for the direction in which the loss falls (Section 1.4), and automatic differentiation, which lets the framework compute the derivatives for us (Section 1.5). Section 1.6 is about doubt: when a measured difference is larger than sampling noise, and how evidence updates a probability. Finally, Section 1.7 shows how to find out what a function does when the book does not say.

This is a long chapter, and a reader at home with NumPy and calculus can skim much of it. We would still read Section 1.5 and the part of Section 1.6 on the standard error, since the rest of the book uses both constantly. Each section develops only what later chapters need. The Mathematics for Deep Learning appendix treats linear algebra (Chapter 33), calculus (Chapter 34), and probability (Chapter 36) in depth, with proofs; readers new to these subjects can turn to it at any point.

Resources and Further Reading

The resources are grouped into books, courses, and documentation, and each names the section it accompanies. All are freely accessible online.

Books

- Python for Data Analysis, 3rd ed. — Wes McKinney — an open-access guide to NumPy and pandas by the creator of pandas; a companion to Section 1.1 and Section 1.2.
- Python Data Science Handbook — Jake VanderPlas — an example-driven tour of NumPy, pandas, and the wider scientific-Python stack used throughout the book.
- Mathematics for Machine Learning — Deisenroth, Faisal & Ong — a development of linear algebra, calculus, and probability aimed at machine learning, in more depth than Section 1.3, Section 1.4, and Section 1.6.
- Introduction to Probability — Blitzstein & Hwang — a free PDF of the textbook of Harvard's Stat 110, whose lecture videos and materials are also online; a fuller treatment of Section 1.6.

Courses and video lectures

- Essence of Linear Algebra — 3Blue1Brown — animated lectures on vectors and matrices as geometric transformations; a visual companion to Section 1.3.
- Essence of Calculus — 3Blue1Brown — animated lectures on limits, derivatives, and the chain rule; a visual companion to Section 1.4 and Section 1.5.
- NumPy: the absolute basics for beginners — NumPy project — the official introduction to the array operations of Section 1.1.
- NumPy Tutorials — NumPy project — runnable notebooks that apply array programming to concrete data tasks.

Tutorials, notes, and documentation

- Array programming with NumPy — Harris, Millman, van der Walt et al. (Nature, 2020) — the reference paper on the n-dimensional array model that every tensor library in this book follows (preprint at arXiv:2006.10256).
- NumPy user guide — NumPy project — the official reference for array creation, indexing, broadcasting, and data types (Section 1.1).
- pandas user guide — pandas project — the official reference for the loading, indexing, and missing-data handling of Section 1.2.
- PyTorch documentation — PyTorch project — the official API reference for tensor operations and automatic differentiation; Section 1.7 lists the corresponding references for JAX, TensorFlow, and MXNet and shows how to use them.
- SciPy documentation — SciPy project — routines for linear algebra, optimization, and statistics beyond what the tensor libraries provide.

1.1 Data Manipulation

A photograph of a shirt reaches a model as a grid of numbers, one per pixel, and everything else the model touches is numbers too: its parameters, the values it computes along the way, and the gradients used to train it. A deep learning program spends nearly all of its time computing with such collections of numbers. How should we store them, and how can we compute with them quickly?

Let us take one humble computation, the rescaling of images, to see what is at stake. Training a model processes its examples in small groups called *minibatches*, and a minibatch of 256 such photographs, grayscale and of 28×28 pixels each, holds $256 \times 28 \times 28 = 200{,}704$ numbers. Image files store each pixel as an integer from 0 to 255, and before a model sees the images we rescale every pixel to a real number between 0 and 1 by dividing it by 255. In plain Python, this takes three nested loops over the images, the rows, and the columns, and the interpreter carries out the 200,704 divisions one at a time. In deep learning code it takes one expression, `images / 255`, which the library applies to the whole minibatch at once in compiled code, on a CPU or a GPU. This is how deep learning code computes throughout: with whole arrays, and hardly ever with loops over their elements.

An array of numbers arranged along any number of axes is called a *tensor*. If you know NumPy (Harris et al. 2020), much of this section will look familiar, and you may read it quickly. The tensor classes of deep learning libraries (`ndarray` in MXNet, `Tensor` in PyTorch and TensorFlow, `jax.Array` in JAX) resemble NumPy's `ndarray`, with two additions, and it is these that make deep learning practical. They run their operations on GPUs and other accelerators as well as on CPUs, and the libraries can compute derivatives of computations on them automatically (Section 1.5).

This section shows how to create tensors and inspect their shape and type, how to rearrange, select, and change their elements, and how to compute with whole tensors at once, including tensors whose shapes differ. It ends with what these computations cost in memory and how to convert tensors to other Python objects. The minibatch of images is our running example. Where a tensor lives, in main memory or on a GPU, changes no result in this chapter; a later chapter shows how to choose.

We import PyTorch, whose package is called `torch`.

```
import torch
```

1.1.1 Tensor Creation

1.1.1.1 Shape

A tensor arranges its numbers along its *axes*. A vector has one axis and a matrix has two. The minibatch of images has four: one for the image, one for the color channel (a grayscale image has a single channel), and one each for the rows and the columns of pixels. Axes are numbered from 0, so for the minibatch, axis 0 indexes the images and axis 3 the columns. The number of axes is the *order* of a tensor (`ndim` in code), and the minibatch has order 4.

The simplest way to create a tensor is `arange(n)`, which returns a vector of the integers from 0 up to, but not including, n. We ask for 32-bit floating-point numbers, the type that we compute with throughout:

```
x = torch.arange(12, dtype=torch.float32)
x
```

```
tensor([ 0.,  1.,  2.,  3.,  4.,  5.,  6.,  7.,  8.,  9., 10., 11.])
```

The numbers in a tensor are its *elements*, and the number of elements is its *size*:

PyTorch calls the size `numel`, for "number of elements".

```
x.numel()
```

```
12
```

The *shape* of a tensor lists its length along each axis. The shape of a vector has one entry, which equals the size:

```
x.shape
```

```
torch.Size([12])
```

Figure 1.1 shows tensors with zero to three axes. A single number, a *scalar*, is a tensor with no axes; its shape is empty. The size of every tensor is the product of the entries of its shape. For the minibatch of images, whose shape is $(256, 1, 28, 28)$, this is the count of 200,704 from the opening, since the single channel contributes a factor of 1.

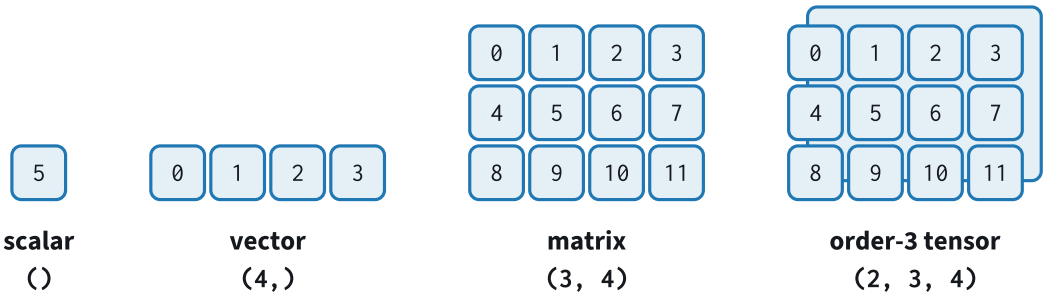


Figure 1.1: Tensors with zero, one, two, and three axes. The shape lists the length of each axis, and the size is the product of these lengths: 1, 4, 12, and 24 elements.

1.1.1.2 Data Types

Every tensor also has a *data type*, its `dtype`, which fixes how each element is stored. Deep learning computes mostly with 32-bit floating-point numbers, `float32`, which take 4 bytes each and keep about 7 significant decimal digits. That is fewer digits than a pocket calculator shows, and it turns out to be enough. Other data need other types. The pixels of an image are 8-bit unsigned integers, `uint8`, which take one byte each and hold the integers from 0 to 255. The data type determines how much memory a tensor needs: the minibatch of images takes 200,704 bytes as `uint8` and four times as much, about 0.8 MB, as `float32`. Large models also compute with 16-bit floating-point numbers, which halve the memory again at some cost in precision (Section 5.5).

With integer arguments, `torch.arange` creates a tensor of 64-bit integers, hence the `float32` in the `arange` cell above. The method `to` converts a tensor to another type, as in `x.to(torch.int32)`.

1.1.1.3 Initial Values

A model needs tensors with initial values before training can change them. The simplest choice is a constant: the functions `zeros` and `ones` create a tensor of a given shape filled with 0s or with 1s. Here is a tensor of zeros with the shape $(2, 3, 4)$ of Figure 1.1:

```
torch.zeros((2, 3, 4))
```

```
tensor([[[0., 0., 0., 0.],
         [0., 0., 0., 0.],
         [0., 0., 0., 0.]],

        [[0., 0., 0., 0.],
         [0., 0., 0., 0.],
         [0., 0., 0., 0.]]])
```

Most parameters of a model, however, must start at small random values. The reason has to wait until Section 4.4; for now we only need to know how to draw them. The following tensor holds numbers drawn independently from the standard normal distribution, a bell-shaped distribution centered at 0 that places about 95% of its draws between -2 and 2:

```
torch.randn(3, 4)
```

```
tensor([[-0.9435,  0.8533,  0.1923,  1.7390],
        [-0.3445, -1.3464,  0.2704,  1.2691],
        [-0.4866,  0.0408,  0.6379,  0.4052]])
```

Random numbers differ from one run to the next, which is a nuisance when we hunt for a bug or want others to reproduce a result. The numbers are, of course, not random at all: a generator computes them from a starting value, the *seed*, and the same seed gives the same numbers. Below, `a` and `c` are the first draws from the same seed and agree, whereas `b` is a different draw:

PyTorch keeps one global generator, which every draw advances, and `torch.manual_seed` resets it.

```
torch.manual_seed(42)
a = torch.randn(3)
b = torch.randn(3)  # The generator has moved on
torch.manual_seed(42)
c = torch.randn(3)  # Same seed: c repeats a
a, b, c
```

```
(tensor([0.3367, 0.1288, 0.2345]),
 tensor([ 0.2303, -1.1229, -0.1863]),
 tensor([0.3367, 0.1288, 0.2345]))
```

Most experiments in this part of the book deliberately do not fix the seed. Every training run starts from freshly drawn weights and visits its data in a new random order, so its result is one draw from a distribution, and the numbers on your screen will differ a little from those printed here. This is why we report many results as a range, such as "between 82% and 84%": it is the spread that we saw when we repeated the experiment. A single number with four digits would claim more than one run can know. Our conclusions are meant to hold anywhere in such a range, and yours should too.

Finally, we can write down the elements of a tensor ourselves, as nested Python lists in which the outermost list runs along axis 0:

```
torch.tensor([[2, 1, 4, 3], [1, 2, 3, 4], [4, 3, 2, 1]])
```

```
tensor([[2, 1, 4, 3],
        [1, 2, 3, 4],
        [4, 3, 2, 1]])
```

This is all we need to build the running example. Until we load a real dataset, random integers from 0 to 255 stand in for the pixels. Such images show static rather than shirts, but only the shape and the type of the minibatch matter here:

```
# 256 grayscale images of 28 x 28 pixels, each pixel an integer in [0, 255]
images = torch.randint(0, 256, (256, 1, 28, 28), dtype=torch.uint8)
images.shape, images.dtype
```

```
(torch.Size([256, 1, 28, 28]), torch.uint8)
```

The shape reads: 256 images, 1 channel, 28 rows, and 28 columns.

1.1.2 Reshaping and Indexing

1.1.2.1 Reshaping

The same numbers often need a different arrangement. A classifier that treats each image as a list of 784 pixel values needs the minibatch as a matrix with one image per row, of shape $(256, 784)$. The function `reshape` changes the shape of a tensor but neither its size nor the order of its elements. On a small example, we turn the vector `x` of shape $(12,)$ into a matrix `X` of shape $(3, 4)$. The elements fill the matrix one row at a time, so `x[3]` becomes `X[0, 3]` (Figure 1.2):

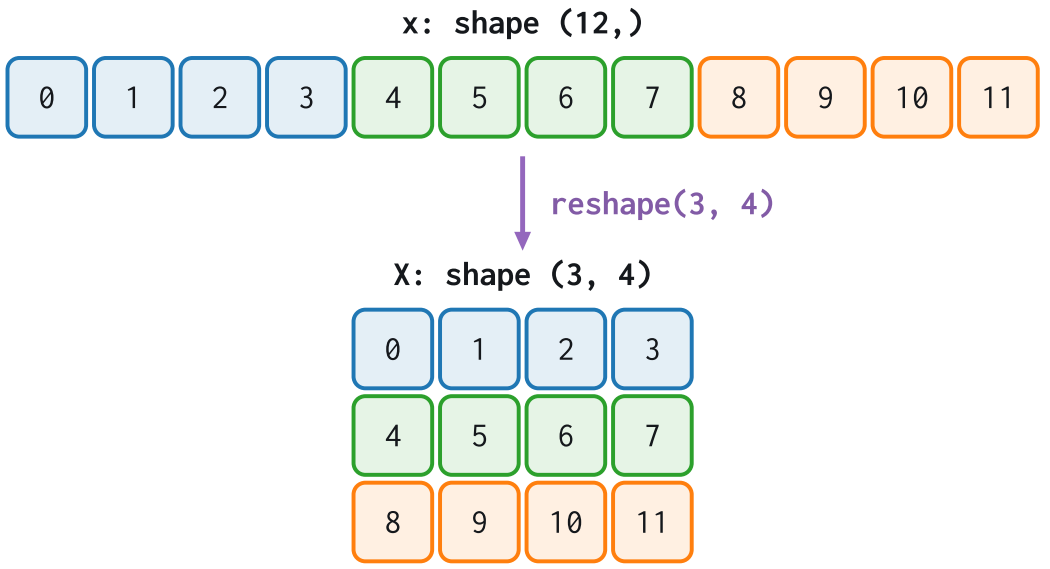


Figure 1.2: Reshaping a vector of 12 elements into a 3×4 matrix. The elements keep their order and fill the matrix one row at a time.

```
X = x.reshape(3, 4)
X
```

```
tensor([[ 0.,  1.,  2.,  3.],
        [ 4.,  5.,  6.,  7.],
        [ 8.,  9., 10., 11.]])
```

Since the size is fixed, one entry of the target shape follows from the others: 12 elements in 3 rows make 4 columns. Writing -1 for one entry lets the library infer it, so `x.reshape(-1, 4)` and `x.reshape(3, -1)` give the same matrix as `x.reshape(3, 4)`. In the same way, we flatten every image of the minibatch into a row of 784 pixels without writing out 784:

```
images.reshape(256, -1).shape
```

```
torch.Size([256, 784])
```

Reshaping can also add or remove an axis of length 1, which leaves the elements and their order unchanged. The first image of the minibatch, `images[0]`, has shape $(1, 28, 28)$. A model that expects minibatches needs it as a minibatch of one image, of shape $(1, 1, 28, 28)$, while a plot needs a 28×28 matrix without the channel axis. The index `None` inserts an axis of length 1, and `squeeze` removes one:

The method `image.unsqueeze(0)` inserts the axis as well.

```
image = images[0]
image[None].shape, image.squeeze(0).shape
```

```
(torch.Size([1, 1, 28, 28]), torch.Size([28, 28]))
```

1.1.2.2 Indexing and Slicing

As with Python lists, we select elements of a tensor by *indexing*. Indices start at 0, and negative indices count from the end. A *slice* `start:stop` selects the positions from `start` up to, but not including, `stop`, and a colon alone selects all positions. Indices for several axes are separated by commas, and an index given alone applies to axis 0. Thus `X[-1]` selects the last row of `X`, and `X[1:3]` selects the second and third rows:

```
X[-1], X[1:3]
```

```
(tensor([ 8.,  9., 10., 11.]),
 tensor([[ 4.,  5.,  6.,  7.],
         [ 8.,  9., 10., 11.]]))
```

Note that the two results differ in shape. An integer index removes its axis, so `X[-1]` is a vector, whereas a slice keeps its axis, so `X[1:3]` is a matrix with two rows (Figure 1.3). The distinction looks pedantic, but many an error about mismatched shapes goes back to it.

On the minibatch, `images[0]` is the first image, `images[:32]` a smaller minibatch of the first 32 images, and `images[:, 0, 14, 14]` the pixel in row 14 and column 14 of every image:

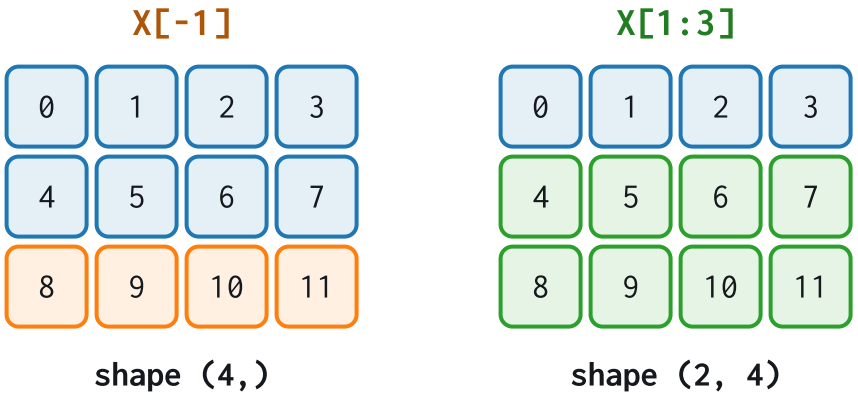


Figure 1.3: Indexing a 3×4 matrix. The integer index `X[-1]` selects the last row and removes axis 0, leaving shape $(4,)$. The slice `X[1:3]` selects rows 1 and 2 and keeps axis 0, leaving shape $(2, 4)$.

```
images[0].shape, images[:32].shape, images[:, 0, 14, 14].shape
```

```
(torch.Size([1, 28, 28]), torch.Size([32, 1, 28, 28]), torch.Size([256]))
```

An index can also be a tensor. Two tensors of integer positions, one for the rows and one for the columns, select one element for each pair of positions. This picks one entry from every row, as when we read off the score that a classifier gives to the correct category of each image. The matrix `X` holds the numbers 0 to 11 row by row, so the rows $(0, 1, 2)$ and the columns $(3, 0, 2)$ read `X[0, 3]`, `X[1, 0]`, and `X[2, 2]`, which are 3, 4, and 10:

```
rows, cols = torch.arange(3), torch.tensor([3, 0, 2])
X[rows, cols]
```

```
tensor([ 3.,  4., 10.])
```

A tensor of booleans with the shape of `X`, a *mask*, selects the elements of `X` where it is `True`. The comparison `X > 2` compares every element of `X` with 2 and gives such a mask, and the operator `&` combines two masks elementwise into one that is `True` where both are (*and*); `|` gives *or*. The selected elements need not form rows and columns, so the result is a vector:

```
mask = (X > 2) & (X < 8)
X[mask]
```

```
tensor([3., 4., 5., 6., 7.])
```

To keep the shape instead, `where(mask, A, B)` takes the elements of `A` where the mask is `True` and those of `B` elsewhere. A single number in place of `A` or `B` stands for that number at every position, so the following sets every element outside the mask to 0:

```
torch.where(mask, X, 0)
```

```
tensor([[0., 0., 0., 3.],
        [4., 5., 6., 7.],
        [0., 0., 0., 0.]])
```

1.1.2.3 Writing Elements

So far we have only read elements. Can we also change them? Here the libraries part ways, by design. MXNet and PyTorch tensors are *mutable*: an assignment overwrites their elements in place. TensorFlow and JAX tensors are *immutable*: once created, they never change.

Writing raises a question that reading could not. When we reshaped `x` into `X`, did the library copy the elements? It need not have: the elements of `x` lie in memory one after another, and so do the rows of `X`, so recording a new shape for the same memory suffices. A tensor that shares its memory with another in this way is called a *view*, and a write through either name is then seen through both. For immutable tensors the question is idle. For mutable ones we had better know the answer.

An index on the left side of an assignment writes the element. We display `x`, from which `X` was reshaped, as well:

```
X[1, 2] = 17
X, x
```

```
(tensor([[ 0.,  1.,  2.,  3.],
         [ 4.,  5., 17.,  7.],
         [ 8.,  9., 10., 11.]]),
 tensor([ 0.,  1.,  2.,  3.,  4.,  5., 17.,  7.,  8.,  9., 10., 11.]))
```

The vector `x` has changed too: its element 6 is now 17, although we never assigned to `x`. PyTorch's `reshape` returned a view. (When the arrangement in memory does not allow a view, `reshape` copies the data instead.)

To give many elements the same value, we index with a slice. Here `[:2, :]` selects the first two rows, with the colon taking all columns:

```
X[:2, :] = 12
X
```

```
tensor([[12., 12., 12., 12.],
        [12., 12., 12., 12.],
        [ 8.,  9., 10., 11.]])
```

1.1.3 Operations

1.1.3.1 Elementwise Operations

An operation is *elementwise* if it applies a function of numbers to each element of a tensor separately, or to each pair of elements at the same position in two tensors of the same shape. The arithmetic operators `+`, `-`, `*`, `/`, and `**` (power) all work this way. We try them on two new vectors, reusing the names `x` and `y`:

```
x = torch.tensor([1.0, 2, 4, 8])
y = torch.tensor([2.0, 2, 2, 2])
x + y, x - y, x * y, x / y, x ** y
```

```
(tensor([ 3.,  4.,  6., 10.]),
 tensor([-1.,  0.,  2.,  6.]),
 tensor([ 2.,  4.,  8., 16.]),
 tensor([0.5000, 1.0000, 2.0000, 4.0000]),
 tensor([ 1.,  4., 16., 64.]))
```

Each result has the shape of the operands, and each element depends only on the two elements at its position: `x / y` halves every element of `x`, and `x ** y` squares it. In symbols, a function f of two numbers turns two tensors $\mathbf{u}$ and $\mathbf{v}$ of the same shape into the tensor $\mathbf{c}$ with elements $c_i = f(u_i, v_i)$, where i runs over all positions.

Functions of a single number, such as e^x, apply to each element in the same way:

```
torch.exp(x)
```

```
tensor([2.7183e+00, 7.3891e+00, 5.4598e+01, 2.9810e+03])
```

We are now in a position to carry out the rescaling from the beginning of the section. Dividing the minibatch by 255 divides every pixel and returns a new tensor. Division with `/` always returns floating-point numbers, here `float32`, even where a quotient happens to be an integer. (Addition makes no such promise: the sum of two `uint8` tensors is again `uint8`, with consequences that an exercise explores.) Strictly speaking, the definition above does not cover `images / 255`, since the second operand is a single number rather than a tensor of the same shape. We return to this case in the next subsection:

```
scaled = images / 255
print(scaled.dtype, images[0, 0, 0, :4], scaled[0, 0, 0, :4], sep='\n')
```

```
torch.float32
tensor([102, 121, 210, 214], dtype=torch.uint8)
tensor([0.4000, 0.4745, 0.8235, 0.8392])
```

The type has changed from `uint8` to `float32`, and the first four pixels of the first image, integers from 0 to 255, have become numbers between 0 and 1.

Comparisons, such as `X > 2` above, are elementwise too. They return a tensor of booleans that is `True` where the comparison holds:

```
x == y
```

```
tensor([False,  True, False, False])
```

Not every operation is elementwise. Summing all the elements of a tensor gives a tensor with a single element. Summing a tensor of booleans counts its `True` entries, because `True` counts as 1 and `False` as 0; this is how we will count the correct predictions of a classifier:

```
x.sum(), (x == y).sum()
```

```
(tensor(15.), tensor(1))
```

A sum can also run along a single axis and leave, for instance, one sum per row of a matrix. Such reductions along an axis are the subject of Section 1.3.2.2.

1.1.3.2 Broadcasting

Elementwise operations are simple when the two tensors have the same shape. Unfortunately, they rarely do. Model code constantly combines a minibatch, stored with one example per row, with a vector that holds one number per column: preprocessing, for instance, subtracts the same vector of column means from every example. Copying the vector once for every example, only to match the shape of the minibatch, would waste memory and time. *Broadcasting*, a convention that deep learning libraries adopted from NumPy, applies the vector to every row without copying it. Here we add the vector $(100, 200, 300, 400)$ to each row of a new 3×4 matrix `X`, which again holds the numbers 0 to 11:

```
X = torch.arange(12, dtype=torch.float32).reshape(3, 4)
v = torch.tensor([100.0, 200, 300, 400])
X + v
```

```
tensor([[100., 201., 302., 403.],
        [104., 205., 306., 407.],
        [108., 209., 310., 411.]])
```

Every row received the same four offsets, one per column.

Broadcasting follows a simple rule, and it is worth knowing by heart. Line the two shapes up from the *right* and compare them axis by axis: two axes are compatible when they are equal or when one of them is 1 (a missing leading axis counts as 1). Each axis of length 1 is then *stretched* to match the other, by virtually copying its entries, and the elementwise operation is applied to the results. If any pair of axes is incompatible, the operation raises an error rather than guessing. For `X + v`, the shapes $(3, 4)$ and $(4,)$ line up as $(3, 4)$ and $(1, 4)$, so the single row of `v` is stretched to three rows. The division of the minibatch by 255 was the simplest case: a scalar has no axes, so it is stretched along all of them.

Both operands can be stretched at once. In Figure 1.4, a column of shape $(3, 1)$ and a row of shape $(1, 2)$ are each stretched to shape $(3, 2)$:

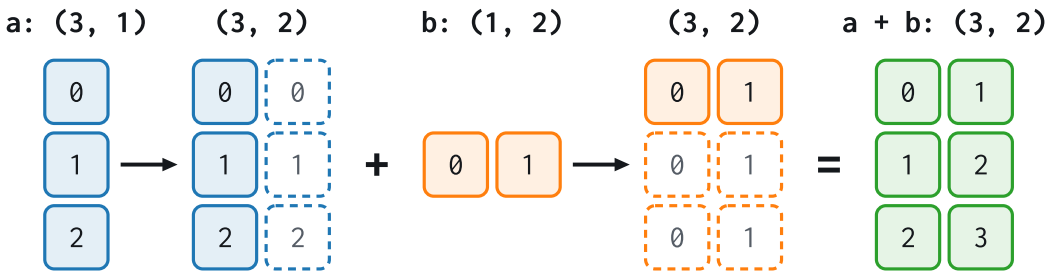


Figure 1.4: Broadcasting stretches axes of length 1. The column `a` of shape $(3, 1)$ and the row `b` of shape $(1, 2)$ are both expanded to shape $(3, 2)$ (dashed: virtual copies) before the elementwise addition.

```
a = torch.arange(3).reshape(3, 1)
b = torch.arange(2).reshape(1, 2)
a, b
```

```
(tensor([[0],
         [1],
         [2]]),
 tensor([[0, 1]]))
```

Their sum adds every element of the column to every element of the row:

```
a + b
```

```
tensor([[0, 1],
        [1, 2],
        [2, 3]])
```

What if the shapes are not compatible? Lining up $(3, 2)$ and $(2, 3)$ from the right pairs 2 with 3 and 3 with 2; in both pairs the lengths differ and neither is 1, so the library refuses:

```
try:
    torch.ones((3, 2)) + torch.ones((2, 3))
except Exception as e:
    print(e)
```

```
The size of tensor a (2) must match the size of tensor b (3) at non-singleton dimension 1
```

The rule also predicts what happens on the minibatch of images. Subtracting a mean image of shape $(28, 28)$ from the minibatch of shape $(256, 1, 28, 28)$ works: the last two axes agree, and the mean image is stretched along the first two. Adding a vector of 256 numbers, one per image, to the flattened minibatch of shape $(256, 784)$ fails, because lining up the shapes from the right pairs 256 with 784. Reshaping the vector to shape $(256, 1)$ repairs this, since $(256, 1)$ and $(256, 784)$ are compatible.

1.1.3.3 Concatenation

Broadcasting combines tensors position by position. *Concatenation* instead joins tensors end to end along an existing axis. Joining two minibatches of 256 images along axis 0 gives one minibatch of 512 images; joining two sets of features for the same examples along axis 1 gives every example more features. The lengths along all other axes must agree. We concatenate two 3×4 matrices, `X` (defined afresh) and `Y`, in both ways. Along axis 0, the result has $3 + 3 = 6$ rows; along axis 1, it has $4 + 4 = 8$ columns:

```
X = torch.arange(12, dtype=torch.float32).reshape(3, 4)
Y = torch.tensor([[2.0, 1, 4, 3], [1, 2, 3, 4], [4, 3, 2, 1]])
torch.cat((X, Y), dim=0), torch.cat((X, Y), dim=1)
```

```
(tensor([[ 0.,  1.,  2.,  3.],
         [ 4.,  5.,  6.,  7.],
         [ 8.,  9., 10., 11.],
         [ 2.,  1.,  4.,  3.],
```

```
        [ 1.,  2.,  3.,  4.],
        [ 4.,  3.,  2.,  1.]]),
 tensor([[ 0.,  1.,  2.,  3.,  2.,  1.,  4.,  3.],
        [ 4.,  5.,  6.,  7.,  1.,  2.,  3.,  4.],
        [ 8.,  9., 10., 11.,  4.,  3.,  2.,  1.]]))
```

Stacking joins tensors of the same shape along a new axis instead. This is how a minibatch arises from single examples: stacking 256 images of shape $(1, 28, 28)$ gives the minibatch of shape $(256, 1, 28, 28)$. Stacking `X` and `Y` along a new axis 0 gives shape $(2, 3, 4)$, and along a new last axis, shape $(3, 4, 2)$:

```
torch.stack((X, Y)).shape, torch.stack((X, Y), dim=2).shape
```

```
(torch.Size([2, 3, 4]), torch.Size([3, 4, 2]))
```

1.1.4 Memory and Conversion

1.1.4.1 Allocation and In-Place Updates

Most operations return a new tensor, which needs new memory. The statement `Y = Y + X` computes the sum in newly allocated memory and then binds the name `Y` to the new tensor. Python's `id` function, which returns a number that identifies an object for as long as the object exists, shows that `Y` now names a different object:

```
before = id(Y)
Y = Y + X
id(Y) == before
```

```
False
```

For small tensors this costs little, and we can forget about it. A model with a billion parameters, however, needs 4 GB to store them as `float32` numbers, and training updates all of them at every step. If every update allocated new memory for its result, the program would need room for the old and the new values at the same time, and would spend time on the allocations. An *in-place* update writes the result into the memory that the tensor already occupies. It has a second effect, which is easily overlooked and matters just as much: it reaches every part of the program that refers to the tensor, whereas rebinding one name leaves other names pointing at the old values. In-place updates come with one restriction, whose reason will become clear in Section 1.5: a library that records operations to compute derivatives may still need an old value, and then an in-place update must not overwrite it.

Slice assignment, `Z[:] = <expression>`, writes the result into the existing tensor `Z`. We create `Z` with `zeros_like`, which returns zeros with the shape and type of its argument. After the assignment, `Z` names the same tensor as before, and its elements hold `X + Y`:

```
Z = torch.zeros_like(Y)
before = id(Z)
Z[:] = X + Y
id(Z) == before
```

```
True
```

Slice assignment still computes `X + Y` in a temporary tensor before copying it into `Z`. Augmented assignment avoids the temporary. If we no longer need the old value of `X`, then `X += Y` adds `Y` to `X` in place:

```
before = id(X)
X += Y
id(X) == before
```

```
True
```

`X` is still the same object, and it now holds the sum.

1.1.4.2 Conversion to Other Python Objects

Plotting, pandas, and many scientific libraries take NumPy arrays or Python numbers rather than tensors, so we convert tensors at the boundary between our code and theirs. Whether a conversion copies the data or shares the memory, as a view does, depends on the library.

The method `numpy` converts a tensor on the CPU to a NumPy array, and `torch.from_numpy` converts a NumPy array to a tensor. Neither copies the data: the tensor and the array share their memory, just as `X` and `x` did after the reshape, so an in-place change to one also changes the other.

```
A = X.numpy()
B = torch.from_numpy(A)
type(A), type(B)
```

```
(numpy.ndarray, torch.Tensor)
```

Writing an element of the NumPy array `A` changes `X` and `B` as well, because all three use the same memory:

```
A[0, 0] = 100  # Write to the NumPy array
X[0, 0], B[0, 0]
```

```
(tensor(100.), tensor(100.))
```

A tensor with a single element converts to a Python number with the method `item` or with the built-in functions `float` and `int`. Printing a summary number such as an average error, or comparing it with a threshold in Python, needs such a conversion. It is not free, though: when the computation runs on a GPU, the conversion has to wait for it to finish, so training code converts as rarely as possible.

```
a = torch.tensor(3.5)
a, a.item(), float(a), int(a)
```

```
(tensor(3.5000), 3.5, 3.5, 3)
```

The numbers in this section were made up. Real data arrive in files, often as tables with missing entries and with text in some of the columns, and they are rarely as tidy as the output of `arange`. The next section turns such a table into tensors.

1.1.5 Summary

Deep learning code stores its data, parameters, and intermediate results in tensors, arrays with a shape and a data type. It computes with whole tensors rather than with loops over their elements: one expression divides all 200,704 pixels of a minibatch by 255. Reshaping, indexing, and concatenation rearrange, select, and join elements. Elementwise operations apply a function of numbers at every position, and broadcasting combines tensors of different shapes by lining up their shapes from the right and stretching axes of length 1. Most operations allocate a new tensor; where memory matters, libraries with mutable tensors update them in place, and libraries with immutable tensors reuse memory through mutable variables or through their compiler.

1.1.6 Exercises

1. code **Inferred reshape axis.** A minibatch of n grayscale images of 28×28 pixels arrives as a tensor of shape $(n, 1, 28, 28)$. Each image must be flattened into a row of 784 values. Check every prediction below in code.
    1. Predict the result of reshaping the minibatch to $(-1, 784)$ and to $(256, -1)$, once for a full minibatch of $n = 256$ images and once for a final minibatch of $n = 100$ images. State which of the two targets works for every minibatch size.
    2. Suppose the images were loaded with three color channels by mistake, so that the minibatch has shape $(n, 3, 28, 28)$. Predict the result of the reshape to $(-1, 784)$, and state whether this step detects the mistake.
2. **Shifted slices.** A tensor `tokens` of shape $(4, 11)$ holds four sequences of *tokens*, integers that stand for words, one sequence per row. A model reads the first 10 tokens of each row and predicts, at every position, the token that follows. Write slice expressions for the input tensor and the target tensor, each of shape $(4, 10)$, so that the target at position t is the token that follows the input at position t. Then give expressions for both tensors that stay correct without change when each row holds $T + 1$ tokens and the model reads the first T, for every $T \geq 1$.
3. code **Broadcasting shapes.** Broadcasting aligns two shapes from the right.
    1. Replace `a` and `b` with two tensors that each have three axes, whose shapes differ on every axis and that still broadcast. Predict the shape of `a + b`, then run the code and check your prediction.
    2. A model returns predictions `y_hat` of shape $(5, 1)$ for five examples whose labels `y` form a vector of shape $(5,)$. Predict the shape of `y_hat - y`. Then let `y` hold the labels $0, 1, 2, 3, 4$ as floating-point numbers and set `y_hat` to `y` reshaped to $(5, 1)$, so that every prediction equals its label. Compute the mean of the squared entries of `y_hat - y`, explain the value, and give the reshape of `y` that pairs each prediction with its own label.
4. code **Limits of data types.** The pixels of `images` are stored as `uint8`, and the rest of the book computes in `float32`.
    1. Predict the result of adding 100 to a `uint8` tensor that holds the value 200, and check your prediction. Explain the result, and brighten the first image of the minibatch by 100 intensity levels without this error by converting its type first.
    2. A `float32` number stores 24 significant binary digits. Explain why `float32` represents every integer up to $2^{24} = 16{,}777{,}216$, and check that adding 1 to 2^{24} in

`float32` leaves it unchanged. A training loop counts the examples it has processed in a `float32` tensor. After how many examples does the count stop growing, and which data type should the counter have instead?

5. code **Masks on the minibatch.** The pixels of `images` are random integers from 0 to 255, so about half of them exceed 127.
   1. With one comparison and one sum, count the pixels of the minibatch that exceed 127, and compare the count with half of the size of `images`. Explain why summing a mask counts.
   2. Use `where` to make a thresholded copy of `images` in which every pixel above 127 becomes 255 and every other pixel 0. Confirm that the copy has the shape of `images` and holds only these two values.
   3. `images[images > 127]` starts from the same mask. Predict its shape, check, and explain why a mask used as an index must return a tensor with one axis, whatever the shape of `images`, while `where` keeps the shape. Name a task for which each form is the right one. Then select the bright pixels of the first image alone, with a slice and a mask.
6. code **In-place composition.** ▣ Compute `(A + B) * (-A / 2)` for two same-shaped tensors without allocating any intermediate tensor, using only in-place methods such as `add_`, `div_`, `neg_`, and `mul_`, or the `out=` argument of the corresponding functions. Confirm that `A.data_ptr()` and `B.data_ptr()` are unchanged and that the result equals the direct expression. Explain why `Z[:] = A + B` would not have met the requirement.

   Adapted from Nicolas P. Rougier, 100 numpy exercises, problem 35.
7. code **Loops against broadcasting.** ▣ For a vector `a` of 200 elements and a vector `b` of 300 elements, compute the table of all absolute differences $d_{ij} = |a_i - b_j|$, of shape $(200, 300)$.
   1. Compute the table once with two nested Python loops, and once with a single broadcast expression after reshaping `a` to $(200, 1)$ and `b` to $(1, 300)$. Confirm that the two results agree, time both, and report the speedup.
   2. The broadcast expression copies neither input. How many bytes does its result take in `float32` when both vectors have 10,000 elements? What does this say about the memory that broadcasting can create?

1.2 Data Preprocessing

The tensors of the previous section were made by our own code, filled with zeros, ones, or random numbers in the shape we chose. Data from the world rarely arrive that way. A table of records is usually stored as a text file, with gaps where a value was not recorded and with words where a model needs numbers. Turning such a table into tensors is this section's subject, and it takes more judgment than the word *preprocessing* suggests.

Our example is a table of ten home sales, each recorded with the number of rooms, the type of roof, the floor area, and the sale price. Small as it is, it has everything that makes real tables awkward. Three of the ten homes lack the number of rooms, and four lack the roof type. The roof type is a word, such as `Slate` or `Tile`, not a number. And the floor area, measured in square feet, is hundreds of times larger than the number of rooms. Before a model sees this table we have to decide what to put in the gaps, how to turn words into numbers, and whether to bring the columns to a common scale.

These decisions change what a model can learn. Dropping every incomplete row would throw away more than half of the table. Numbering slate roofs, tile roofs, and missing entries 0, 1, and 2 would invent an order among them that does not exist. And left unscaled, the floor area dominates any computation that compares homes, so that the number of rooms barely counts. This section turns the table into a tensor of inputs and a tensor of targets with the pandas library (McKinney 2010), the standard Python tool for loading, inspecting, and transforming tables (Figure 1.5).

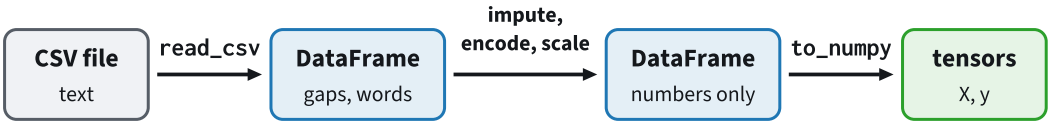


Figure 1.5: From a file to tensors. `read_csv` loads the text file into a DataFrame with gaps and text columns; imputation, encoding, and scaling make every entry a number; `to_numpy` hands the result to the framework as the tensors `X` and `y`.

We import pandas under its usual alias `pd`, the module `os` for file paths, and the deep learning framework, which we need only at the end, for the conversion to tensors.

```
import os
import pandas as pd
import torch
```

1.2.1 Loading and Inspecting

1.2.1.1 CSV Files

A *comma-separated values* (CSV) file stores a table as plain text. Each line holds one row, the first line usually holds the column names, and commas separate the fields within a line. We write the ten home sales to the file `../data/house_tiny.csv`, with the columns `NumRooms`, `RoofType`, `Area` (in square feet), and `Price` (in dollars). An empty field marks a missing entry.

```
os.makedirs(os.path.join('..', 'data'), exist_ok=True)
data_file = os.path.join('..', 'data', 'house_tiny.csv')
with open(data_file, 'w') as f:
    f.write('''NumRooms,RoofType,Area,Price
3,Slate,1500,210000
,Tile,2100,290000
2,,850,127500
4,Slate,1940,258000
,,1200,168000
3,Tile,1650,225000
5,Slate,2600,375000
2,,900,142000
,Tile,1750,240000
4,,2050,295000''')
```

The function `read_csv` parses the file into a *DataFrame*, the table object of pandas. A DataFrame consists of named columns, each holding values of one type, and it labels its rows; the numbers 0 to 9 at the left of the output are these labels.

```
data = pd.read_csv(data_file)
data
```

```
   NumRooms RoofType  Area   Price
0       3.0    Slate  1500  210000
1       NaN     Tile  2100  290000
2       2.0      NaN   850  127500
3       4.0    Slate  1940  258000
4       NaN      NaN  1200  168000
5       3.0     Tile  1650  225000
6       5.0    Slate  2600  375000
7       2.0      NaN   900  142000
8       NaN     Tile  1750  240000
9       4.0      NaN  2050  295000
```

Every empty field has become `NaN` (not a number), the marker that pandas uses for a *missing value*.

1.2.1.2 Data Types and Summaries

Before transforming a table, it pays to check what pandas made of it: how many rows and columns it has, how many entries of each column are present, and which data type pandas inferred for each column. The method `info` reports all three.

```
data.info()
```

```
<class 'pandas.DataFrame'>
RangeIndex: 10 entries, 0 to 9
Data columns (total 4 columns):
 #   Column    Non-Null Count  Dtype
---  ------    --------------  -----
 0   NumRooms  7 non-null      float64
 1   RoofType  6 non-null      str
 2   Area      10 non-null     int64
 3   Price     10 non-null     int64
dtypes: float64(1), int64(2), str(1)
memory usage: 481.0 bytes
```

The table has 10 rows and 4 columns. `NumRooms` has 7 entries present and `RoofType` 6, so 3 and 4 entries are missing. The data types show how pandas read each column. `Area` and `Price` hold 64-bit integers. `NumRooms` holds whole numbers too, and yet pandas has made it a `float64` column. The gaps are to blame: `NaN` is a floating-point value, and an integer column cannot hold it. `RoofType` has the type `str`, which pandas uses for text (versions before pandas 3.0 used `object`). A text column usually holds categories, as here, and must be turned into numbers before a model can use it.

For the numerical columns, the method `describe` summarizes the values: how many are present, their mean and standard deviation, and their quantiles.

```
data.describe()
```

```
       NumRooms         Area          Price
count  7.000000    10.000000      10.000000
mean   3.285714  1654.000000  233050.000000
std    1.112697   556.421104   76128.710455
min    2.000000   850.000000  127500.000000
25%    2.500000  1275.000000  178500.000000
50%    3.000000  1700.000000  232500.000000
75%    4.000000  2022.500000  282000.000000
max    5.000000  2600.000000  375000.000000
```

The columns have very different scales. The number of rooms ranges from 2 to 5, the area from 850 to 2,600 square feet, and the price from 127,500 to 375,000 dollars. The floor area averages 1,654 square feet, about 500 times the average number of rooms, 3.3. We return to this difference when we rescale the columns.

1.2.1.3 Inputs and Targets

We want to predict the price of a home from its other attributes. The price is the *target*, also called the *label*, and the remaining columns are the inputs, also called *features*. We separate the two first, so that the steps below transform only the inputs and the price never ends up among them. The indexer `iloc` selects by position, with the slicing rules of tensors: `data.iloc[:, :3]` takes all rows and the first three columns, and `data.iloc[:, 3]` takes the fourth column. The names of the selected columns confirm the split:

```
inputs, targets = data.iloc[:, :3], data.iloc[:, 3]
list(inputs.columns), targets.name
```

```
(['NumRooms', 'RoofType', 'Area'], 'Price')
```

Selection by position is short, but it breaks without an error when a new version of the file inserts or reorders columns. Selection by name, as in `data.drop(columns='Price')`, does not; an exercise at the end of the section takes this up.

1.2.2 Cleaning and Encoding

1.2.2.1 Missing Values

Values go missing in real data for many reasons: a field was not recorded, did not apply, or was lost. Two methods count the gaps. `isna` returns a table of the same shape that holds `True` where an entry is missing and `False` elsewhere, and `sum` adds up each column, counting `True` as 1.

```
inputs.isna().sum()
```

```
NumRooms    3
RoofType    4
Area        0
dtype: int64
```

There are three common responses, and each gives something up.

- *Deletion* drops every row, or every column, that contains a missing value. It is simple, but it discards data. If values are missing more often for some kind of home, such as small ones, deletion also biases what remains.
- *Imputation* fills each gap with an estimate, such as the mean of the column. It keeps every row, but the filled values are guesses, and they make the column look less spread out than it is.
- An *indicator* is an extra column that records which values were missing, usually added alongside imputation. With it, a model can learn whether a missing value carries information of its own, for example when the number of rooms is missing mostly for large homes. The cost is one more column per feature, which a small table can ill afford.

Let us see what deletion would do to our table. Dropping every row that has a missing entry keeps only the complete rows:

```
len(inputs.dropna()), len(inputs)
```

```
(4, 10)
```

Only 4 of the 10 homes remain, far too few, so we impute instead. For a numerical column, the simplest estimate is the column mean; the median is the safer choice when a few very large values pull the mean away from the typical value. The method `fillna` fills the gaps of each column with the value given for that column, and `mean(numeric_only=True)` computes the mean of each numerical column, skipping its missing entries. A text column has no mean, so the gaps in `RoofType` remain; we deal with them when we encode the column.

```
inputs = inputs.fillna(inputs.mean(numeric_only=True))
inputs
```

```
   NumRooms RoofType  Area
0  3.000000    Slate  1500
1  3.285714     Tile  2100
2  2.000000      NaN   850
3  4.000000    Slate  1940
4  3.285714      NaN  1200
5  3.000000     Tile  1650
6  5.000000    Slate  2600
7  2.000000      NaN   900
8  3.285714     Tile  1750
9  4.000000      NaN  2050
```

The three missing room counts are now 3.29, the mean $23/7$ of the seven observed counts. No home has 3.29 rooms, of course. An imputed value is a placeholder, and we meet these three again when we rescale the column.

1.2.2.2 Categorical Features

The column `RoofType` holds *categories*: each home belongs to one of a few groups, and the groups have no order. Counting a missing entry as a group of its own, there are three groups: `Slate`, `Tile`, and missing. A model needs numbers, and the numbering 0, 1, and 2 is the first thing that comes to mind. Unfortunately, it does more than invent an order. It also places

a missing roof twice as far from slate as tile is, as though not knowing the roof were an extreme case of tile.

One-hot encoding avoids both defects. It replaces the column by one new column per category, which holds 1 for the homes in that category and 0 for all others, so that any two categories are equally far apart. The function `get_dummies` performs the encoding. With `dummy_na=True`, it treats a missing value as one more category, which is the indicator strategy applied to a categorical column. With `dtype=float`, the new columns hold the numbers 0.0 and 1.0 rather than the values `False` and `True`.

```
inputs = pd.get_dummies(inputs, dummy_na=True, dtype=float)
inputs
```

	NumRooms	Area	RoofType_Slate	RoofType_Tile	RoofType_nan
0	3.000000	1500	1.0	0.0	0.0
1	3.285714	2100	0.0	1.0	0.0
2	2.000000	850	0.0	0.0	1.0
3	4.000000	1940	1.0	0.0	0.0
4	3.285714	1200	0.0	0.0	1.0
5	3.000000	1650	0.0	1.0	0.0
6	5.000000	2600	1.0	0.0	0.0
7	2.000000	900	0.0	0.0	1.0
8	3.285714	1750	0.0	1.0	0.0
9	4.000000	2050	0.0	0.0	1.0

`RoofType` has become the three columns `RoofType_Slate`, `RoofType_Tile`, and `RoofType_nan`, and each row has a single 1 among them. One of the three is therefore redundant, and for a column with only two values a single 0/1 column suffices. The numerical columns pass through unchanged, so every column of `inputs` now holds numbers.

One-hot encoding suits columns with a few categories. A column with thousands of distinct values, such as a postal code, would become thousands of columns that are almost entirely zero. Such columns are better represented by *embeddings*, short vectors of numbers that a model learns for each value; we meet them in Section 8.4.

1.2.3 Scaling and Converting

1.2.3.1 Standardization

The two numerical columns still measure different things in different units. In raw units, two homes with the same area that differ by two rooms are closer to each other than two homes with the same number of rooms that differ by three square feet. Any computation that adds up differences between features, such as the distance between two homes, is then decided almost entirely by the area. A feature on a much larger scale than the others also slows down training, for a reason that Section 2.3 gives.

The standard remedy is *standardization*. We replace every entry x of a column by

$$\frac{x - \mu}{\sigma}, \tag{1.1}$$

where μ is the mean of the column and σ its standard deviation, the typical distance of an entry from the mean. For a column with entries $x_1, \dots, x_n$, pandas computes it as $\sigma =$

$\sqrt{\frac{1}{n-1}\sum_{i=1}^{n}(x_i-\mu)^2}$. (Dividing by $n-1$ instead of n is a convention that Section 1.6 explains.) After this step every column has mean 0 and standard deviation 1, and an entry says how many standard deviations it lies above or below the mean of its column. The column no longer remembers whether it was measured in square feet or in square meters. The one-hot columns already share a scale, 0 and 1, so we leave them as they are.

```
numerical = ['NumRooms', 'Area']
mu, sigma = inputs[numerical].mean(), inputs[numerical].std()
inputs[numerical] = (inputs[numerical] - mu) / sigma
inputs
```

```
   NumRooms      Area  RoofType_Slate  RoofType_Tile  RoofType_nan
0 -0.314485 -0.276769             1.0            0.0           0.0
1  0.000000  0.801551             0.0            1.0           0.0
2 -1.415185 -1.444949             0.0            0.0           1.0
3  0.786214  0.513999             1.0            0.0           0.0
4  0.000000 -0.815929             0.0            0.0           1.0
5 -0.314485 -0.007189             0.0            1.0           0.0
6  1.886913  1.700151             1.0            0.0           0.0
7 -1.415185 -1.355089             0.0            0.0           1.0
8  0.000000  0.172531             0.0            1.0           0.0
9  0.786214  0.711691             0.0            0.0           1.0
```

The largest home, in row 6, has 5 rooms and 2,600 square feet. After standardization these entries are 1.89 and 1.70: in both columns the home lies 1.7 to 1.9 standard deviations above the mean, and the two numbers can now be compared directly. The imputed room counts have become 0, because mean imputation placed them at the mean of the column. The placeholders have left a trace, however. Imputation shrank the standard deviation of `NumRooms` from 1.11, that of the seven observed counts, to 0.91. The three imputed entries sit at the mean and add no spread, so every standardized room count moves farther from 0. With the observed spread, the largest home would score 1.54 instead of 1.89.

One shortcut of this section should not be copied. All statistics, the means used for imputation and the means and standard deviations used for standardization, were computed from the whole table, with no homes held out. In practice we hold back part of the data to test a trained model, and the test has to imitate data that the model has never seen. The statistics must therefore be computed from the training data alone and then applied, unchanged, to the test data. Computing them from all the data lets information about the test data *leak* into training, and the test then overstates how well the model will do on new data. Section 4.7 follows this rule on a real dataset of home prices.

1.2.3.2 Conversion to Tensors

What remains is a change of container. The method `to_numpy` converts a DataFrame to a NumPy array, and the framework's tensor constructor converts the array to a tensor. A DataFrame stores floating-point columns as 64-bit numbers, whereas deep learning frameworks compute in 32-bit floating point by default, so we ask `to_numpy` for `float32` directly.

```
X = torch.tensor(inputs.to_numpy(dtype='float32'))
y = torch.tensor(targets.to_numpy(dtype='float32'))
```

```
X.shape, X.dtype, y.shape, y.dtype
```

```
(torch.Size([10, 5]), torch.float32, torch.Size([10]), torch.float32)
```

`X` has one row per home and one column per feature: the two standardized numerical features and the three indicators of the roof type. `y` holds the ten prices.

Ten homes are, admittedly, a toy. The pandas library holds a whole table in main memory and processes it on the CPU, which suits the tables in this book. Tables larger than memory need tools that read them in pieces, and images, audio, and text need pipelines of their own, which later chapters build. The tensor `X` is a matrix with one row per example, the form in which models receive their data. What they do with such a matrix is the subject of the next section.

1.2.4 Summary

Real tables have missing entries, text columns, and columns on very different scales, whereas a model needs a tensor of numbers. The pandas library loads such a table into a DataFrame, and `info` and `describe` show its size, its gaps, its data types, and the ranges of its columns. Missing values can be deleted, imputed, or flagged with an indicator column; on small data, imputation keeps the rows that deletion would discard. Categorical columns become one 0/1 column per category, which avoids inventing an order, and numerical columns are standardized to mean 0 and standard deviation 1, with statistics computed from the training data only. `to_numpy` then hands the table to the framework as a `float32` tensor.

1.2.5 Exercises

1. code **Indexing by name.** This section splits `data` by position with `data.iloc[:, :3]` and `data.iloc[:, 3]`. Suppose a new version of `house_tiny.csv` adds a column `YearBuilt` between `Area` and `Price`.
    1. State what the positional split then puts into `inputs` and `targets`, and whether any later line of the section's code raises an error.
    2. Rewrite the split with column names so that `targets` holds `Price` and `inputs` holds every other column, for the new file and for the original one. The pandas user guide on indexing lists the options.
2. code **Imputation strategies.** Fill the missing values in `NumRooms` in three ways: with the column mean, with the column median, and with the column mean plus a "was-missing" indicator column.
    1. For the mean fill and the median fill, compute the mean and the standard deviation of the filled column, compare them with those of the observed values, and explain the direction of each change.
    2. Suppose that in a larger dataset `NumRooms` is missing mostly for large homes. State what a model sees for those homes after mean imputation alone. Then explain how a model that predicts the price as a weighted sum of its features plus a constant can use the indicator column to correct its predictions for them.
    3. State what each of the three treatments assumes about why the values are missing.

Adapted from Kaggle Learn, Intermediate Machine Learning, Exercise 2.

3. **High-cardinality categories.** One-hot encoding turns a categorical column with k distinct values into k columns, plus one more with `dummy_na=True`.

    1. A table of $n = 10^6$ homes has a `ZipCode` column with no missing entries and $k = 10^4$ distinct values. Compute how many bytes its one-hot columns occupy when stored densely as `float64`, and how many of their entries are nonzero.
    2. A column `ListingID` holds a value unique to each row. Consider three treatments: one-hot encoding, an embedding (a learned vector for each distinct value), and dropping the column. For each, state what a model could learn about a home whose `ListingID` does not occur in the training data, and choose one treatment.

    Adapted from Kaggle Learn, Intermediate Machine Learning, Exercise 3.

4. code **Where leakage hides.** This section fills the missing `NumRooms` entries with the mean of all observed values and standardizes `NumRooms` and `Area` with the mean and standard deviation of all ten rows. Treat the first eight rows of `data` as training data and the last two as test data.

    1. Redo the imputation and the standardization with statistics computed on the training rows only and applied unchanged to the test rows. Compare the standardized test rows with those of the whole-dataset recipe.
    2. Explain what the whole-dataset recipe lets the model learn about the test rows that the training-only recipe does not.
    3. Suppose a feature is constant on the training rows but not on the test rows. State what the training-only recipe produces for that feature on each split, and propose a fix.

5. [extended] **A real messy dataset, end to end.** Take a messy public dataset, for example the San Francisco restaurant-inspection data used in Berkeley's Data 100: one row per inspected business, with its name, address, zip code, latitude and longitude, and the scores of its health inspections, where an unknown latitude or longitude is recorded as -9999 rather than as a blank field. Run the full pipeline of this section: detect such a non-standard missing-value sentinel, choose and justify a handling strategy, encode at least one categorical column with an explicit judgment about its number of distinct values, standardize the numerical columns, and convert the result to a tensor. Report the shape and data type of the final tensor.

    Adapted from UC Berkeley Data 100, homework 2A "Food Safety".

6. code **Memory per row.** `data.info(memory_usage='deep')` reports the memory that the table occupies, text included. Compute the bytes per row of `data` as loaded, of `inputs` after imputation and one-hot encoding, and of the final `float32` tensor `X`. Which column costs most per row as loaded, and why? Then estimate how many homes a table of this kind could hold in 16 GB of main memory at each of the three stages.

1.3 Linear Algebra

Most operations of Section 1.1 act on each element of a tensor separately: dividing a mini-batch by 255 rescales every pixel on its own. A model has to do more. To decide what an image shows, it combines many numbers into a few, weighing some pixels more than others. Linear algebra studies such weighted sums. It lets us write the computation of a model

on a whole minibatch as one expression, and it tells us, before we run anything, what shape the result has and how much work it takes.

Our example is a simple classifier for small grayscale images of clothing, like those of the Fashion-MNIST dataset that later chapters use. It sorts them into ten categories such as shirts, sneakers, and bags. Each image has $28 \times 28 = 784$ pixels. The classifier gives every category a score, a weighted sum of the 784 pixel intensities plus an offset, and predicts the category with the highest score. Training scores a minibatch of 256 images at a time (Section 1.1), thousands of times over, so Python loops over the numbers are out of the question.

In the notation of linear algebra, the scores take one line. Stack the images as the rows of a matrix $\mathbf{X}$, the weights of each category as the columns of a matrix $\mathbf{W}$, and the ten offsets (also called biases) in a vector $\mathbf{b}$. Together, $\mathbf{W}$ and $\mathbf{b}$ hold the $784 \times 10 + 10 = 7{,}850$ parameters of the classifier. The $256 \times 10 = 2{,}560$ scores of the minibatch are then

$$\mathbf{O} = \mathbf{X}\mathbf{W} + \mathbf{b}, \tag{1.2}$$

where $\mathbf{b}$ is added to every row (Figure 1.6). A library evaluates the expression with a single call to a routine tuned for the hardware.

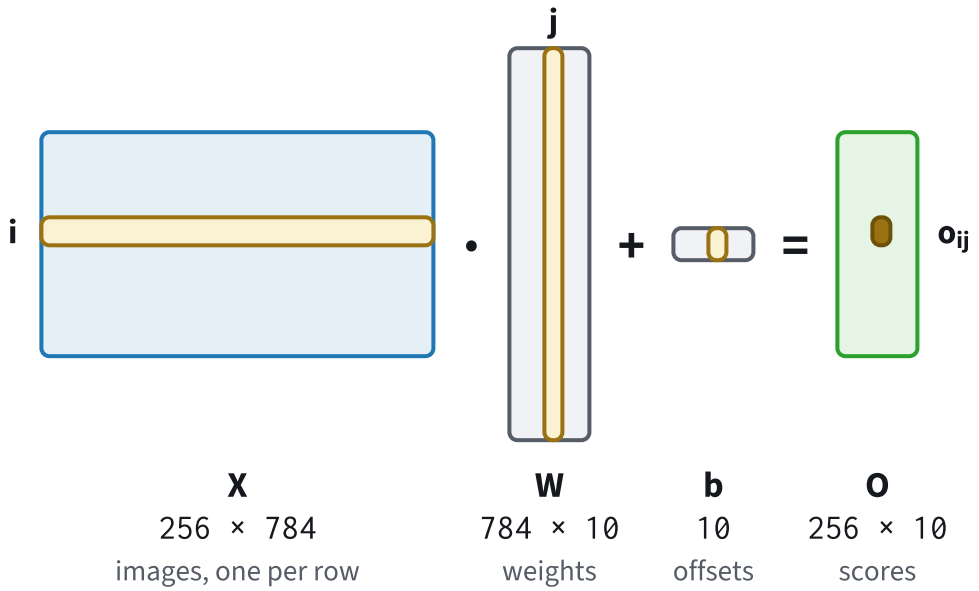


Figure 1.6: The scores of a linear classifier for a minibatch of 256 images and 10 categories. Row i of $\mathbf{X}$ holds image i, column j of $\mathbf{W}$ holds the weights of category j, and the two meet in the score o_{ij}.

We have not yet said what a matrix is, let alone a product of two. This section builds the expression piece by piece and then asks two questions that every model raises: how large is a vector, such as a list of prediction errors, and what happens to a vector when a deep network multiplies it by one matrix after another. Section 1.1 introduced tensors as arrays in code; here they acquire their mathematical meaning. The book's notation is collected in Notation.

```
import torch
```

1.3.1 Scalars, Vectors, Matrices, and Tensors

The objects of linear algebra differ in how many indices it takes to pick out one of their numbers: none for a scalar, one for a vector, two for a matrix, and more for a general tensor. In code, this count is the number of axes of the tensor that stores the object.

1.3.1.1 Scalars

A single number is a scalar: one pixel intensity, the offset of one category, or the price of one home. We denote scalars by lowercase letters such as x and y, and the set of real numbers by $\mathbb{R}$; the expression $x \in \mathbb{R}$, read "x in $\mathbb{R}$", states that x is a real number. In code, a scalar is a tensor with one element and no axes, so its shape is the empty tuple:

```
x = torch.tensor(3.0)
x, x.shape
```

```
(tensor(3.), torch.Size([]))
```

1.3.1.2 Vectors

A fixed-length list of scalars is a *vector*, and the scalars are its elements (also called entries or components). Flattening one image into its 784 pixel intensities gives a vector, and so do the 784 weights that one category assigns to the pixels. We denote vectors by bold lowercase letters such as $\mathbf{x}$ and $\mathbf{w}$, and write $\mathbf{x} \in \mathbb{R}^n$ for a vector of n real elements. By default we write a vector as a column,

$$\mathbf{x} = \begin{bmatrix} x_1 \\ \vdots \\ x_n \end{bmatrix}, \tag{1.3}$$

where the scalar x_i is its i^{th} element. In running text we write the same column vector as $\mathbf{x} = (x_1, \ldots, x_n)$, and we list the entries of a matrix row the same way, as in "the rows $(0, 1, 2)$ and $(3, 4, 5)$". In code, a vector is a tensor with one axis:

```
x = torch.arange(3, dtype=torch.float32)
x
```

```
tensor([0., 1., 2.])
```

Mathematics numbers the elements from 1, whereas Python numbers them from 0. The third element x_3 is therefore `x[2]`:

```
x[2]
```

```
tensor(2.)
```

The number of elements n is the *dimensionality* of the vector. In code it is the length of the tensor, and the only entry of its `shape`:

```
len(x), x.shape
```

```
(3, torch.Size([3]))
```

This is not the number of axes, which Section 1.1 called the order of a tensor (`ndim` in code): a vector in $\mathbb{R}^{784}$ has order 1 and dimensionality 784. The word "dimension" is used loosely for both, and so shall we.

1.3.1.3 Matrices

A table of scalars with m rows and n columns is a *matrix*, written $\mathbf{A} \in \mathbb{R}^{m \times n}$. A minibatch of 256 flattened images is a matrix $\mathbf{X} \in \mathbb{R}^{256 \times 784}$ with one example per row and one feature, the intensity of one pixel position, per column; the book uses this layout throughout. We denote matrices by bold capital letters and write a_{ij} for the element in row i and column j:

$$\mathbf{A} = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{bmatrix}. \tag{1.4}$$

A matrix with as many rows as columns is *square*. In code, a matrix is a tensor with two axes, and `reshape` arranges a vector's elements into one. The 2×3 matrix $\mathbf{A}$ below serves as a small example for the rest of the section:

```
A = torch.arange(6, dtype=torch.float32).reshape(2, 3)
A
```

```
tensor([[0., 1., 2.],
        [3., 4., 5.]])
```

Exchanging the rows and columns of a matrix gives its *transpose*. If $\mathbf{B} = \mathbf{A}^\top$, then $b_{ij} = a_{ji}$ for all i and j, so the transpose of an $m \times n$ matrix is an $n \times m$ matrix:

$$\mathbf{A}^\top = \begin{bmatrix} a_{11} & a_{21} & \cdots & a_{m1} \\ a_{12} & a_{22} & \cdots & a_{m2} \\ \vdots & \vdots & \ddots & \vdots \\ a_{1n} & a_{2n} & \cdots & a_{mn} \end{bmatrix}.$$

The classifier stores the weights of one category per column of $\mathbf{W}$; its transpose $\mathbf{W}^\top$ holds the same weights with one category per row.

```
A.T
```

```
tensor([[0., 3.],
        [1., 4.],
        [2., 5.]])
```

A square matrix that equals its transpose is *symmetric*. Symmetric matrices record symmetric relations, such as the similarity between every pair of examples. They are also the best-behaved matrices we will meet, in a sense that the end of this section makes precise. The following matrix **S** is symmetric:

```
S = torch.tensor([[2.0, 1.0], [1.0, 2.0]])
(S == S.T).all()
```

```
tensor(True)
```

1.3.1.4 Tensors

Images do not arrive as rows. A minibatch of 256 grayscale images is a tensor of shape $(256, 1, 28, 28)$ and order 4. To feed the classifier, we flatten each image into a row of 784 pixels. Random numbers between 0 and 1 stand in for real images whose pixels have already been rescaled to $[0, 1]$, as in Section 1.1. Only the shapes matter here:

```
images = torch.rand(256, 1, 28, 28)
X = images.reshape(256, -1)  # One row of 784 pixels per image
images.shape, X.shape
```

```
(torch.Size([256, 1, 28, 28]), torch.Size([256, 784]))
```

1.3.2 Elementwise Operations and Reductions

The classifier adds the offsets **b** to every row of **XW**, and training averages a measure of error over all the examples in a minibatch. The first is an elementwise operation; the second is a *reduction*, which summarizes many numbers by one.

1.3.2.1 Elementwise Operations

Elementwise operations and broadcasting work as described in Section 1.1. The elementwise product of two matrices of the same shape is called their *Hadamard product* and written $\odot$: $[\mathbf{A} \odot \mathbf{B}]_{ij} = a_{ij} b_{ij}$. It is not the matrix product that appears in **XW**, which we define below. In code, `*` computes the Hadamard product, so `A * A` squares every entry of the 2×3 matrix **A**, whereas the matrix product of **A** with itself is not even defined:

```
A * A
```

```
tensor([[ 0.,  1.,  4.],
        [ 9., 16., 25.]])
```

1.3.2.2 Reductions

The sum of a vector, $\sum_{i=1}^{n} x_i$, and the sum of all elements of a matrix, $\sum_{i=1}^{m} \sum_{j=1}^{n} a_{ij}$, are reductions; so are the mean, the maximum, and the average error over a minibatch. Often we reduce along one axis only. Summing a matrix over its row index i (axis 0) leaves one sum per column, $\sum_i a_{ij}$, a vector of length n. Summing over its column index j (axis 1) leaves one sum per row, a vector of length m. Either way, the axis that we reduce over disappears from the shape (Figure 1.7).

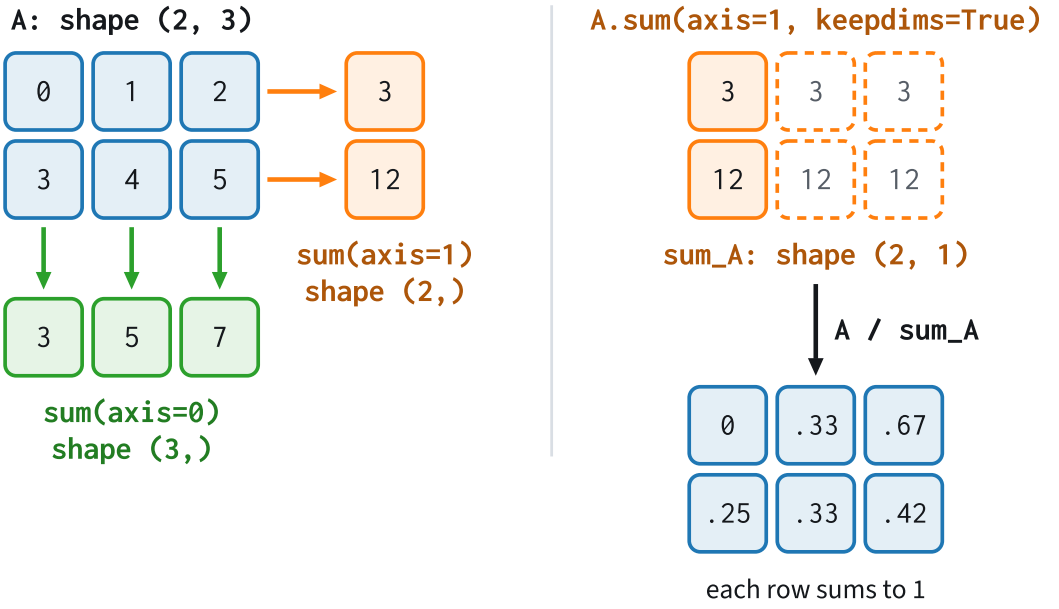


Figure 1.7: Reductions of a 2×3 matrix. Summing over axis 0 leaves one sum per column, and summing over axis 1 leaves one sum per row; the reduced axis disappears from the shape. With `keepdims=True`, the row sums keep the shape $(2, 1)$, so broadcasting stretches them across the three columns and the division `A / sum_A` scales every row to sum to 1.

The argument `axis` selects the axis to reduce; without it, the sum runs over all elements:

```
A.sum(), A.sum(axis=0), A.sum(axis=1)
```

```
(tensor(15.), tensor([3., 5., 7.]), tensor([ 3., 12.]))
```

The mean, the maximum, and the other reductions take the same `axis` argument.

1.3.2.3 Keeping the Reduced Axis

A reduction often feeds back into an elementwise operation with the tensor it came from. Dividing each row of a nonnegative matrix by the row's sum turns the row into proportions that add up to 1, which is the last step when a classifier turns its scores into probabilities. The row sums of $\mathbf{A}$ have shape $(2,)$, however, and broadcasting lines up shapes from the right (Section 1.1.3.2). It would pair the two row sums with the three columns of $\mathbf{A}$, and fail. The argument `keepdims=True` keeps the reduced axis with length 1. The row sums then form a 2×1 column, which broadcasting stretches across the columns (Figure 1.7):

```
sum_A = A.sum(axis=1, keepdims=True)
sum_A, sum_A.shape
```

```
(tensor([[ 3.],
         [12.]]),
 torch.Size([2, 1]))
```

PyTorch's own names for the arguments `axis` and `keepdims` are `dim` and `keepdim`. Its reductions accept both spellings, and the code in this book uses both.

Each row of the quotient now sums to 1:

```
A / sum_A
```

```
tensor([[0.0000, 0.3333, 0.6667],
        [0.2500, 0.3333, 0.4167]])
```

On the minibatch $\mathbf{X}$, subtracting from each image its mean intensity needs one mean per row, and `keepdims=True` gives the 256 means the shape $(256, 1)$ that broadcasts against $(256, 784)$. Subtracting the mean of each pixel position, one per column, needs no `keepdims`, because the 784 column means have shape $(784,)$ and line up with the last axis:

```
col_means = X.mean(axis=0)
row_means = X.mean(axis=1, keepdims=True)
col_means.shape, row_means.shape, (X - row_means).shape
```

```
(torch.Size([784]), torch.Size([256, 1]), torch.Size([256, 784]))
```

For a square matrix, forgetting `keepdims` raises no error and subtracts the wrong numbers, as Exercise 3 shows.

1.3.3 Products

Elementwise operations combine numbers at the same position, and reductions summarize them. A score needs both: multiply each pixel by its weight, then add up the products. This combination is the dot product, and all products of vectors and matrices are built from it.

1.3.3.1 Dot Products

For one image $\mathbf{x} \in \mathbb{R}^{784}$ and the weights $\mathbf{w} \in \mathbb{R}^{784}$ of one category, the score is $\sum_{i=1}^{784} w_i x_i + b$. The sum is the *dot product* of the two vectors. For $\mathbf{x}, \mathbf{y} \in \mathbb{R}^d$ it is

$$\mathbf{x}^\top \mathbf{y} = \sum_{i=1}^{d} x_i y_i. \tag{1.5}$$

The dot product is also called the *inner product* and written $\langle \mathbf{x}, \mathbf{y} \rangle$. It is symmetric, $\mathbf{x}^\top \mathbf{y} = \mathbf{y}^\top \mathbf{x}$, and linear in each argument: $(\alpha \mathbf{x} + \beta \mathbf{z})^\top \mathbf{y} = \alpha\, \mathbf{x}^\top \mathbf{y} + \beta\, \mathbf{z}^\top \mathbf{y}$ for scalars α and β. When $\mathbf{y}$ is the vector of all ones, the dot product is the sum of the elements of $\mathbf{x}$; for the vector $\mathbf{x} = (0, 1, 2)$ from above, it is 3:

```
y = torch.ones(3)
x, y, torch.dot(x, y)
```

```
(tensor([0., 1., 2.]), tensor([1., 1., 1.]), tensor(3.))
```

Equivalently, a dot product is a Hadamard product followed by a sum:

```
torch.sum(x * y)
```

```
tensor(3.)
```

If the weights are nonnegative and sum to 1, the dot product $\mathbf{w}^\top \mathbf{x}$ is a *weighted average* of the elements of $\mathbf{x}$.

1.3.3.2 Matrix–Vector Products

Now score one category for all 256 images of the minibatch. Each score is the dot product of one row of $\mathbf{X}$ with the same weight vector $\mathbf{w}$, and the 256 scores together form a *matrix–vector product*. In general, write a matrix $\mathbf{A} \in \mathbb{R}^{m \times n}$ in terms of its rows $\mathbf{a}_1^\top, \ldots, \mathbf{a}_m^\top$, each a row vector of length n. Its product with a vector $\mathbf{x} \in \mathbb{R}^n$ is the vector of the m dot products,

$$\mathbf{A}\mathbf{x} = \begin{bmatrix} \mathbf{a}_1^\top \\ \mathbf{a}_2^\top \\ \vdots \\ \mathbf{a}_m^\top \end{bmatrix} \mathbf{x} = \begin{bmatrix} \mathbf{a}_1^\top \mathbf{x} \\ \mathbf{a}_2^\top \mathbf{x} \\ \vdots \\ \mathbf{a}_m^\top \mathbf{x} \end{bmatrix} \in \mathbb{R}^m.$$

The product exists only if the number of columns of $\mathbf{A}$ equals the length of $\mathbf{x}$, and it has one element per row of $\mathbf{A}$ (Figure 1.8). For the minibatch, the scores $\mathbf{X}\mathbf{w}+b$ of one category form a vector in $\mathbb{R}^{256}$, one score per image. The same product, fitted to one real-valued label per example such as a price, is *linear regression*, the subject of the next chapter.

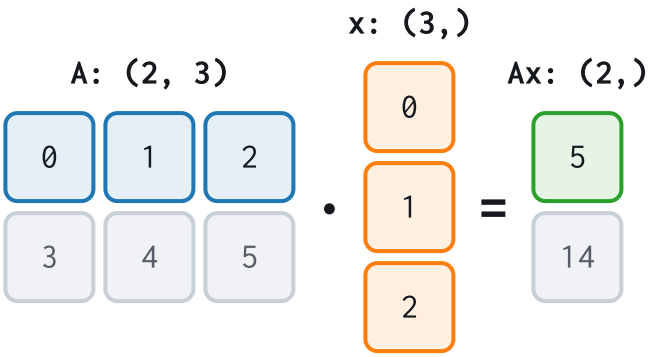


Figure 1.8: The matrix–vector product $\mathbf{A}\mathbf{x}$ takes one dot product per row of $\mathbf{A}$, so a 2×3 matrix maps a vector of length 3 to a vector of length 2. The first row gives $0 \cdot 0 + 1 \cdot 1 + 2 \cdot 2 = 5$.

In code, take the matrix $\mathbf{A}$, with rows $(0, 1, 2)$ and $(3, 4, 5)$, and the vector $\mathbf{x} = (0, 1, 2)$ from above.

The operator `@` computes the matrix–vector product:

```
A.shape, x.shape, A @ x
```

```
(torch.Size([2, 3]), torch.Size([3]), tensor([ 5., 14.]))
```

There is a second way to read the product, column by column, and the last part of this section rests on it. Multiplication by $\mathbf{A} \in \mathbb{R}^{m\times n}$ maps vectors in $\mathbb{R}^n$ to vectors in $\mathbb{R}^m$, and the map is *linear*: $\mathbf{A}(\alpha\mathbf{x}+\beta\mathbf{y}) = \alpha\mathbf{A}\mathbf{x}+\beta\mathbf{A}\mathbf{y}$. What does such a map do to the *coordinate vectors* $\mathbf{e}_j$, which have a 1 in position j and 0 elsewhere? It returns a column: the product $\mathbf{A}\mathbf{e}_j$ is column j of $\mathbf{A}$, and for our 2×3 example $\mathbf{A}\mathbf{e}_2 = (1, 4)$. That settles every other vector as well. Any $\mathbf{x}$ is a weighted sum of coordinate vectors, $\mathbf{x} = \sum_j x_j\mathbf{e}_j$, so by linearity $\mathbf{A}\mathbf{x} = \sum_j x_j\mathbf{A}\mathbf{e}_j$ is the same weighted sum of the columns. For $\mathbf{x} = (0, 1, 2)$ it is $1\cdot(1, 4)+2\cdot(2, 5) = (5, 14)$, the output above. A matrix is thus a list of the places to which it sends the coordinate vectors.

In the plane we can draw the whole map. A 2×2 matrix turns the unit square, spanned by $\mathbf{e}_1$ and $\mathbf{e}_2$, into the parallelogram spanned by its two columns (Figure 1.9). The rotation by $90°$, for instance, sends $\mathbf{e}_1$ to $\mathbf{e}_2$ and $\mathbf{e}_2$ to $-\mathbf{e}_1$, so its columns are $(0, 1)$ and $(-1, 0)$.

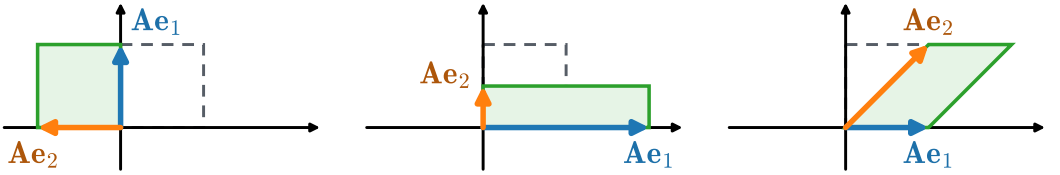


Figure 1.9: A 2×2 matrix maps the unit square (dashed) to the parallelogram spanned by its columns $\mathbf{A}\mathbf{e}_1$ and $\mathbf{A}\mathbf{e}_2$. From left to right: a rotation by $90°$, a scaling by 2 along $\mathbf{e}_1$ and by $1/2$ along $\mathbf{e}_2$, and a shear.

1.3.3.3 Matrix–Matrix Products

Scoring all ten categories takes one matrix–vector product per category, that is, per column of $\mathbf{W}$. Placing the ten results side by side gives the *matrix–matrix product* $\mathbf{X}\mathbf{W}$, usually called simply the matrix product. In general, the product of $\mathbf{A} \in \mathbb{R}^{m\times k}$ and $\mathbf{B} \in \mathbb{R}^{k\times n}$ is the matrix $\mathbf{C} = \mathbf{A}\mathbf{B} \in \mathbb{R}^{m\times n}$ whose entry c_{ij} is the dot product of row i of $\mathbf{A}$ with column j of $\mathbf{B}$:

$$c_{ij} = \sum_{l=1}^{k} a_{il}\, b_{lj}. \tag{1.6}$$

Column j of $\mathbf{C}$ is the matrix–vector product of $\mathbf{A}$ with column j of $\mathbf{B}$. The inner sizes must agree, and the order of the factors matters: $\mathbf{B}\mathbf{A}$ need not even exist. For the classifier, $(256\times 784)(784\times 10)$ gives 256×10, and score o_{ij} is the dot product of image i with the weights of category j (Figure 1.6). Each column of $\mathbf{W}$ acts as a template of 784 pixel weights, and the classifier scores how well each image matches each template.

A small example first. Every column of $\mathbf{B}$ below is the all-ones vector, so every column of $\mathbf{A}\mathbf{B}$ holds the row sums of $\mathbf{A}$, 3 and 12:

```
B = torch.ones(3, 4)
A @ B
```

```
tensor([[ 3.,  3.,  3.,  3.],
        [12., 12., 12., 12.]])
```

Now the classifier itself, with small random weights in place of trained ones. Its prediction for an image is the category with the largest score, the position of the maximum along axis 1, which `argmax` computes:

```
W = 0.01 * torch.randn(784, 10)  # Column j: category j
b = torch.zeros(10)              # Entry j: category j
O = X @ W + b                    # Row i: image i
O.shape, O.argmax(dim=1).shape
```

```
(torch.Size([256, 10]), torch.Size([256]))
```

What did this cost? Each of the mn entries of $\mathbf{AB}$ is a dot product of length k, which takes k multiplications and $k-1$ additions, so the product takes about $2mkn$ floating-point operations (FLOPs). For the minibatch, that is $2 \times 256 \times 784 \times 10 \approx 4 \times 10^6$ FLOPs. This is modest, but it does not stay so. For square matrices of size n, the work $2n^3$ grows faster than the $3n^2$ numbers involved, and a product of two 4096×4096 matrices already takes about 1.4×10^{11} FLOPs. How fast hardware gets through them is a question for Chapter 13.

Two square matrices have names of their own. The *identity matrix* $\mathbf{I}$, with ones on the diagonal and zeros elsewhere, has the coordinate vectors as its columns and therefore changes nothing: $\mathbf{Ix} = \mathbf{x}$ for every $\mathbf{x}$. The *inverse* $\mathbf{A}^{-1}$ of a square matrix $\mathbf{A}$ is the matrix that undoes it, $\mathbf{A}^{-1}\mathbf{A} = \mathbf{A}\mathbf{A}^{-1} = \mathbf{I}$. Not every matrix has one. The scaling of Figure 1.9 is undone by scaling by $1/2$ and by 2, but a matrix that sends some nonzero vector to zero has lost that vector for good (Section 33.2.2.3). A layer of a neural network, finally, applies a linear map, adds an offset, and then applies a nonlinear function to every element.

1.3.4 Norms

Once a model makes predictions, training must measure how wrong they are, and often also how large the weights have become. Both are questions about the size of a vector: of the weights $\mathbf{w}$ or, for a model that predicts one number per example, such as a price, of the errors $\hat{\mathbf{y}} - \mathbf{y} \in \mathbb{R}^{256}$ of the predictions $\hat{\mathbf{y}}$ against the labels $\mathbf{y}$ of a minibatch. A *norm* answers such a question with one nonnegative number, and a norm of the errors, or its square, is a common choice of *loss*, the quantity that training minimizes. There are several norms, and the choice among them matters: as a loss or as a penalty on the weights, each makes the model learn something different.

1.3.4.1 Length and Angle

The ℓ_2 *norm* of a vector is its Euclidean length. It follows from the theorem of Pythagoras, applied one coordinate at a time, and equals the square root of the dot product of the vector with itself:

$$\|\mathbf{x}\|_2 = \sqrt{\mathbf{x}^\top \mathbf{x}} = \sqrt{\sum_{i=1}^{n} x_i^2}.$$

It is the default norm, so we often write $\|\mathbf{x}\|$ without the subscript. Its square, applied to the errors, is the sum of squared errors; divided by the number of examples, it becomes the mean squared error, the standard loss for predicting real numbers. For $\mathbf{x} = (3, -4)$ the ℓ_2 norm is $\sqrt{9+16} = 5$.

Length also explains why the dot product serves to compare vectors. Take two vectors $\mathbf{x}$ and $\mathbf{y}$ in the plane, at angle θ, and the triangle with sides $\mathbf{x}$, $\mathbf{y}$, and $\mathbf{x}-\mathbf{y}$. The law of cosines states that $\|\mathbf{x}-\mathbf{y}\|^2 = \|\mathbf{x}\|^2 + \|\mathbf{y}\|^2 - 2\|\mathbf{x}\|\|\mathbf{y}\|\cos\theta$. Writing the left side as a dot product and expanding it gives $(\mathbf{x}-\mathbf{y})^\top(\mathbf{x}-\mathbf{y}) = \|\mathbf{x}\|^2 - 2\mathbf{x}^\top\mathbf{y} + \|\mathbf{y}\|^2$ instead. Comparing the two expressions,

$$\mathbf{x}^\top\mathbf{y} = \|\mathbf{x}\|\|\mathbf{y}\|\cos\theta. \tag{1.7}$$

The dot product is thus the length of $\mathbf{x}$ times the signed length $\|\mathbf{y}\|\cos\theta$ of the shadow that $\mathbf{y}$ casts on the line through $\mathbf{x}$ (Figure 1.10). It is positive when the angle between the vectors is less than $90°$, zero when they are perpendicular, and negative when the angle exceeds $90°$.

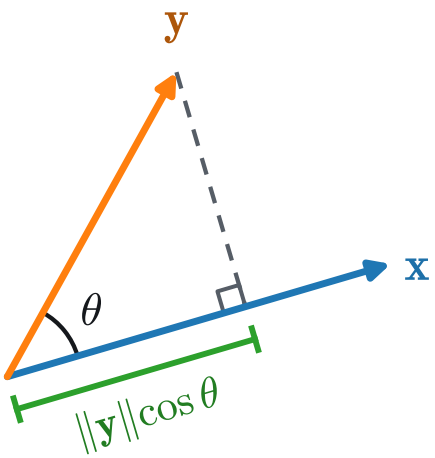


Figure 1.10: The dot product $\mathbf{x}^\top\mathbf{y} = \|\mathbf{x}\|\|\mathbf{y}\|\cos\theta$ is the length of $\mathbf{x}$ times the signed length $\|\mathbf{y}\|\cos\theta$ of the projection of $\mathbf{y}$ onto the line through $\mathbf{x}$.

In 784 dimensions we cannot draw the angle between two images, so we read Equation 1.7 backward and use it to define the angle:

$$\cos\theta = \frac{\mathbf{x}^\top\mathbf{y}}{\|\mathbf{x}\|\|\mathbf{y}\|}. \tag{1.8}$$

This ratio is called the *cosine similarity*. It compares the directions of two vectors and ignores their lengths: an image and a copy of it at twice the brightness, $2\mathbf{x}$, have cosine similarity 1. Two vectors of length 1 at angle θ have dot product $\cos\theta$, which ranges from 1 for vectors that point the same way to -1 for opposite ones (Figure 1.11).

Are we entitled to do this? The definition Equation 1.8 makes sense only if the ratio lies between -1 and 1, as a cosine must. It does, in any number of dimensions.

Proposition (Cauchy–Schwarz inequality). For all $\mathbf{x}, \mathbf{y} \in \mathbb{R}^d$, $|\mathbf{x}^\top\mathbf{y}| \le \|\mathbf{x}\|\|\mathbf{y}\|$.

Proof. If $\mathbf{x}$ or $\mathbf{y}$ is zero, both sides vanish. Otherwise, let $\mathbf{u} = \mathbf{x}/\|\mathbf{x}\|$ and $\mathbf{v} = \mathbf{y}/\|\mathbf{y}\|$, two vectors of length 1. A squared length cannot be negative, so

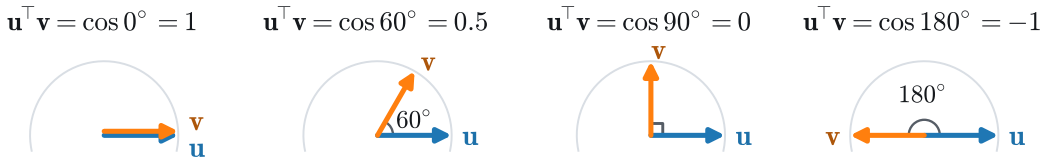


Figure 1.11: Two unit vectors at angle θ: their dot product equals $\cos\theta$, ranging from 1 (aligned) through 0 (perpendicular) to -1 (opposed).

$$0 \le \|\mathbf{u}-\mathbf{v}\|^2 = \|\mathbf{u}\|^2 - 2\mathbf{u}^\top\mathbf{v} + \|\mathbf{v}\|^2 = 2 - 2\mathbf{u}^\top\mathbf{v},$$

which gives $\mathbf{u}^\top\mathbf{v} \le 1$. The same argument applied to $\|\mathbf{u}+\mathbf{v}\|^2$ gives $\mathbf{u}^\top\mathbf{v} \ge -1$. Multiplying both bounds by $\|\mathbf{x}\|\|\mathbf{y}\|$ proves the claim. ■

Equality holds only if $\mathbf{u} = \pm\mathbf{v}$, that is, if one vector is a multiple of the other. In code, the cosine similarity takes one line. The two vectors below have length 2 and dot product 2, so their cosine similarity is $1/2$ and their angle is $\pi/3$, or 60°. Doubling one of them changes nothing:

```
def cosine(x, y):
    return torch.dot(x, y) / (torch.linalg.vector_norm(x) *
                              torch.linalg.vector_norm(y))

x, y = torch.tensor([1.0, 1, 1, 1]), torch.tensor([1.0, 1, 1, -1])
cosine(x, y), cosine(2 * x, y), torch.acos(cosine(x, y))
```

```
(tensor(0.5000), tensor(0.5000), tensor(1.0472))
```

1.3.4.2 Other Vector Norms

The ℓ_2 norm squares the entries before adding them, so a few large entries can outweigh all the small ones. For losses and penalties we sometimes want a measure of size that weighs large and small entries more evenly.

The ℓ_1 *norm* sums the absolute values of the elements,

$$\|\mathbf{x}\|_1 = \sum_{i=1}^{n} |x_i|. \tag{1.9}$$

For $\mathbf{x} = (3, -4)$ it is $3 + 4 = 7$. If the coordinates count city blocks, the ℓ_2 norm is the distance in a straight line and the ℓ_1 norm the distance walked along the streets, hence the name *Manhattan distance* (Figure 1.12).

In code, the ℓ_2 and the ℓ_1 norm of $(3, -4)$ are:

```
x = torch.tensor([3.0, -4.0])
torch.linalg.vector_norm(x), torch.abs(x).sum()
```

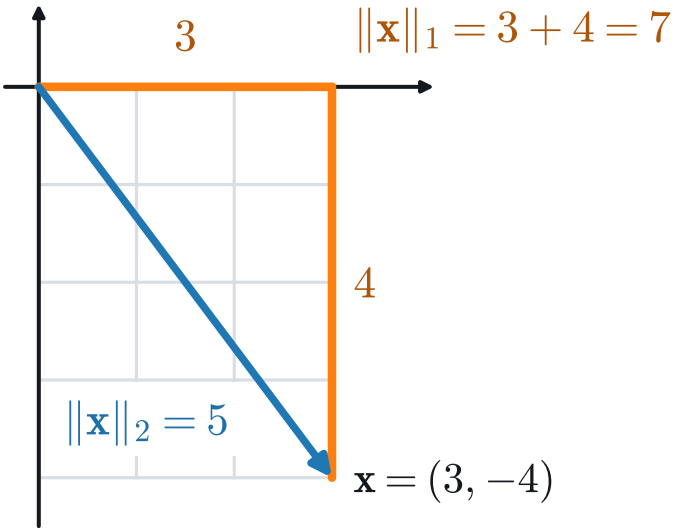


Figure 1.12: The ℓ_2 norm of $\mathbf{x}$ is the length of the straight path to $\mathbf{x}$; the ℓ_1 norm is the length of the path along the grid.

```
(tensor(5.), tensor(7.))
```

In the error vector $(1, -1, 1, -1, 10)$, the one large error contributes $100/104 \approx 96\%$ of the squared ℓ_2 norm but only $10/14 \approx 71\%$ of the ℓ_1 norm. A loss built from squared errors is therefore dominated by a few outliers, while a loss built from absolute errors limits their influence. As penalties on the weights they differ as well: the squared ℓ_2 norm shrinks all weights toward zero, while the ℓ_1 norm tends to set many of them to zero, for reasons that Section 2.7 explains.

Both are special cases of the ℓ_p *norm*, defined for $p \geq 1$ by

$$\|\mathbf{x}\|_p = \left(\sum_{i=1}^{n} |x_i|^p \right)^{1/p}. \tag{1.10}$$

As p grows, the largest element dominates the sum, and in the limit $\|\mathbf{x}\|_\infty = \max_i |x_i|$.

In general, a norm is any function $\|\cdot\|$ from vectors to scalars with three properties:

1. *Scaling*: $\|\alpha\mathbf{x}\| = |\alpha|\|\mathbf{x}\|$ for every scalar α. Doubling a vector doubles its size.
2. *Triangle inequality*: $\|\mathbf{x}+\mathbf{y}\| \leq \|\mathbf{x}\| + \|\mathbf{y}\|$. A detour is never shorter than the direct path.
3. *Positivity*: $\|\mathbf{x}\| > 0$ for every $\mathbf{x} \neq \mathbf{0}$. Only the zero vector has size zero.

For the ℓ_2 norm, the triangle inequality follows from the Cauchy–Schwarz inequality, which bounds the middle term in

$$\|\mathbf{x} + \mathbf{y}\|^2 = \|\mathbf{x}\|^2 + 2\mathbf{x}^\top\mathbf{y} + \|\mathbf{y}\|^2 \leq \|\mathbf{x}\|^2 + 2\|\mathbf{x}\|\|\mathbf{y}\| + \|\mathbf{y}\|^2 = (\|\mathbf{x}\| + \|\mathbf{y}\|)^2. \tag{1.11}$$

The condition $p \geq 1$ is needed. For $p = 1/2$, the vectors $(1, 0)$ and $(0, 1)$ both have size 1, but their sum has size $(1 + 1)^2 = 4 > 2$, which violates the triangle inequality.

1.3.4.3 Matrix Norms

The weights of the classifier form a matrix, so we also need the size of a matrix. The simplest answer treats the matrix as one long vector. The *Frobenius norm* is the ℓ_2 norm of all the entries,

$$\|\mathbf{A}\|_{\mathrm{F}} = \sqrt{\sum_{i=1}^{m} \sum_{j=1}^{n} a_{ij}^2}. \tag{1.12}$$

For the 4×9 matrix of ones it is $\sqrt{36} = 6$:

```
torch.linalg.matrix_norm(torch.ones(4, 9))
```

```
tensor(6.)
```

Weight decay, a common way to keep the weights of a model small, adds a multiple of $\|\mathbf{W}\|_{\mathrm{F}}^2$ to the loss. The Frobenius norm is rather crude in one respect, though. It forgets that a matrix is also a map, and a map raises a different question of size: by how much can it stretch a vector? The rest of the section answers it.

1.3.5 Eigenvectors and Eigenvalues

This part asks more of the reader than the rest of the section, and it can be skipped on a first reading. The book needs it for the analysis of weight decay (Section 2.7) and again in Section 4.4.

A deep network applies one layer after another. Ignore everything but the matrices, and let every layer use the same matrix $\mathbf{A}$, which must be square so that the output of one layer fits the input of the next: after k layers, an input $\mathbf{x}$ has become $\mathbf{A}^k\mathbf{x}$. How large is the result? For a scalar a the answer is a^k, and it is extreme even for values of a close to 1: $1.1^{50} \approx 117$, while $0.9^{50} \approx 0.005$. A 10% deviation from 1 per layer changes the size of a signal by a factor of more than 100, in either direction, after 50 layers. For a matrix, the answer depends on the direction of $\mathbf{x}$, and in a few special directions it reduces to the scalar case.

1.3.5.1 Eigenvectors

A nonzero vector $\mathbf{v}$ is an *eigenvector* of a square matrix $\mathbf{A}$, with *eigenvalue* λ, if

$$\mathbf{A}\mathbf{v} = \lambda\mathbf{v}. \tag{1.13}$$

The matrix does not turn an eigenvector; it only stretches it by the factor λ, and reverses it if $\lambda < 0$. Along $\mathbf{v}$, multiplying by $\mathbf{A}$ is the same as multiplying by the number λ, so $\mathbf{A}^k\mathbf{v} = \lambda^k\mathbf{v}$. In Figure 1.9, the scaling has eigenvectors $\mathbf{e}_1$ and $\mathbf{e}_2$ with eigenvalues 2 and $1/2$, and the shear keeps the direction of $\mathbf{e}_1$, with eigenvalue 1. The rotation turns every direction, so it has no real eigenvector at all.

As a worked example, take the symmetric matrix from above,

$$\mathbf{S} = \begin{bmatrix} 2 & 1 \\ 1 & 2 \end{bmatrix}. \tag{1.14}$$

Its eigenvectors can be found by inspection. Since $\mathbf{S}$ maps $(1, 1)$ to $(3, 3)$ and $(-1, 1)$ to itself, these two directions are eigenvectors, with eigenvalues 3 and 1. Figure 1.13 (left) shows what this means: $\mathbf{S}$ maps the unit circle to an ellipse whose axes lie along the two eigenvectors, with half-lengths 3 and 1. Every other direction is turned toward the eigenvector with the larger eigenvalue; $\mathbf{e}_1 = (1, 0)$, for instance, is mapped to $(2, 1)$.

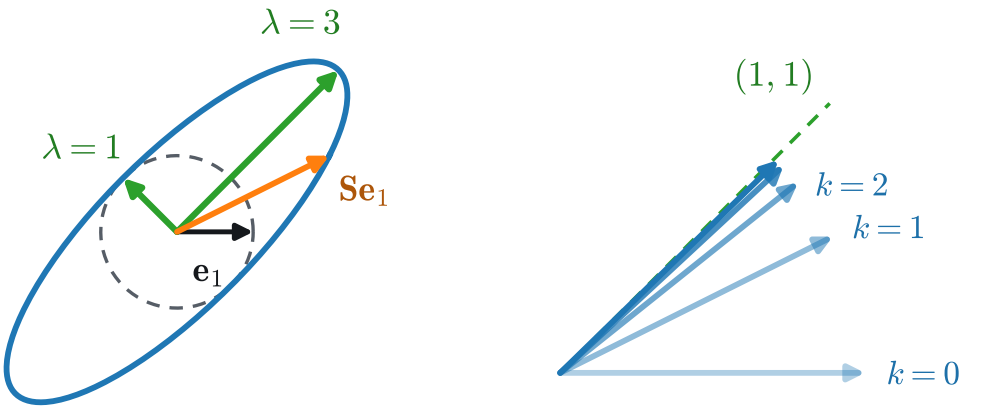


Figure 1.13: Left: $\mathbf{S}$ maps the unit circle (dashed) to an ellipse whose axes are its eigenvectors, stretched by $\lambda = 3$ and $\lambda = 1$, while $\mathbf{e}_1$ is turned to $\mathbf{Se}_1 = (2, 1)$. Right: the directions of $\mathbf{S}^k\mathbf{e}_1$ for $k = 0, \ldots, 4$ approach the eigenvector $(1, 1)$ of the largest eigenvalue.

For symmetric matrices, libraries compute eigenvalues and eigenvectors with `eigh`:

```
S = torch.tensor([[2.0, 1.0], [1.0, 2.0]])
eigenvalues, eigenvectors = torch.linalg.eigh(S)
eigenvalues, eigenvectors
```

```
(tensor([1., 3.]),
 tensor([[-0.7071,  0.7071],
         [ 0.7071,  0.7071]]))
```

The eigenvalues come in increasing order, and the eigenvectors are the columns of the second result, scaled to length 1; their signs are arbitrary, since $-\mathbf{v}$ is an eigenvector whenever $\mathbf{v}$ is. The two eigenvectors of $\mathbf{S}$ are perpendicular. This holds for every symmetric matrix: its eigenvalues are real, and it has a full set of perpendicular eigenvectors, one per dimension (for a proof, see Section 33.3). A matrix that is not symmetric need not have either property, as the rotation shows.

1.3.5.2 Powers of a Matrix

Eigenvectors answer the question posed at the start of Section 1.3.5: how large $\mathbf{A}^k\mathbf{x}$ becomes after k layers. We know what happens to an eigenvector, so the idea is to take $\mathbf{x}$ apart into

eigenvectors. Let us do this first for $\mathbf{S}$ and $\mathbf{x} = \mathbf{e}_1$. Half the difference of the two eigenvectors does it: $\mathbf{e}_1 = \frac{1}{2}(1,1) - \frac{1}{2}(-1,1)$. Every multiplication by $\mathbf{S}$ triples the first part and leaves the second alone, so $\mathbf{S}^k\mathbf{e}_1 = \frac{1}{2}3^k(1,1) - \frac{1}{2}(-1,1) = ((3^k+1)/2,\ (3^k-1)/2)$. The iterates are $(2,1)$, $(5,4)$, $(14,13)$, and so on, and their direction approaches $(1,1)$ (Figure 1.13, right).

Nothing in this computation was special to $\mathbf{S}$. A symmetric $n \times n$ matrix $\mathbf{A}$ has n perpendicular eigenvectors $\mathbf{v}_1, \dots, \mathbf{v}_n$, which we scale to length 1, and since there is one per dimension, any vector is a sum $\mathbf{x} = \sum_i c_i \mathbf{v}_i$. The coefficients are easy to find. Taking the dot product of both sides with $\mathbf{v}_j$ removes every term but one, because the eigenvectors are perpendicular, and leaves $c_j = \mathbf{v}_j^\top \mathbf{x}$. Applying $\mathbf{A}$ to $\mathbf{x}$ k times multiplies each part by its own eigenvalue k times:

$$\mathbf{A}^k\mathbf{x} = \sum_{i=1}^{n} c_i \lambda_i^k \mathbf{v}_i. \tag{1.15}$$

The part with the largest $|\lambda_i|$ grows fastest, so it soon dominates: $\mathbf{A}^k\mathbf{x}$ turns toward the corresponding eigenvector, and its length grows by a factor of about $\max_i |\lambda_i|$ per step. Ten multiplications by $\mathbf{S}$ confirm both predictions:

```
norm = torch.linalg.vector_norm
v = torch.tensor([1.0, 0.0])
for _ in range(10):
    v_prev, v = v, S @ v
growth = norm(v) / norm(v_prev)
growth, v / norm(v)
```

```
(tensor(3.0000), tensor([0.7071, 0.7071]))
```

The length grows by a factor of 3, the largest eigenvalue, per step, and the direction agrees with the eigenvector $(1,1)/\sqrt{2} \approx (0.7071, 0.7071)$ to four decimals. Repeated multiplication, with the vector rescaled to length 1 after each step so that its entries stay in range, is called *power iteration*. It finds the largest eigenvalue in absolute value and its eigenvector using nothing but matrix–vector products, provided that this eigenvalue is strictly larger in absolute value than all others and that the starting vector has a nonzero component along its eigenvector.

The same decomposition answers the question that the Frobenius norm left open. The largest ratio $\|\mathbf{A}\mathbf{x}\|/\|\mathbf{x}\|$ over all $\mathbf{x} \neq \mathbf{0}$ is the *spectral norm* of $\mathbf{A}$. For a symmetric matrix it is the largest $|\lambda_i|$, since $\|\mathbf{A}\mathbf{x}\|^2 = \sum_i c_i^2 \lambda_i^2 \le \max_i \lambda_i^2 \sum_i c_i^2 = \max_i \lambda_i^2 \, \|\mathbf{x}\|^2$, with equality for the eigenvector of that eigenvalue. Matrices that are not symmetric need more care (Section 33.5).

Now return to the deep network. The matrix

$$\mathbf{M} = \begin{bmatrix} 1 & 0.1 \\ 0.1 & 1 \end{bmatrix}$$

has the same eigenvectors as $\mathbf{S}$, with eigenvalues 1.1 and 0.9, both within 10% of 1. We apply it 50 times to both eigenvectors at once, stored as the columns of a matrix, and then

measure the length of each column:

```
M = torch.tensor([[1.0, 0.1], [0.1, 1.0]])
# The eigenvectors (1, 1) and (-1, 1) of M, as columns of length 1
V = torch.tensor([[1.0, -1.0], [1.0, 1.0]]) / 2 ** 0.5
for _ in range(50):  # 50 "layers"
    V = M @ V
torch.linalg.vector_norm(V, dim=0)
```

```
tensor([1.1739e+02, 5.1538e-03])
```

After 50 layers, the part of a signal along the first eigenvector has grown by a factor of about 117, and the part along the second has shrunk to about 0.005 of its size, as 1.1^{50} and 0.9^{50} predict. Real networks use a different matrix in each layer, with nonlinear functions in between, but the effect persists: unless the layers stretch vectors by factors close to 1, signals, and during training gradients, grow or vanish exponentially with depth. Section 4.4 shows how careful initialization keeps these factors close to 1.

Everything in this section took the weights as given, and ours were random numbers. What we cannot do yet is improve the weights. That requires knowing how the error changes when a parameter changes, which is the subject of the next section.

1.3.6 Summary

Linear algebra writes the computation of a model on a whole minibatch as a few operations on vectors and matrices: with one image per row of $\mathbf{X}$, a linear classifier computes all its scores as $\mathbf{O} = \mathbf{X}\mathbf{W} + \mathbf{b}$. Reductions summarize a tensor along an axis, and `keepdims` keeps that axis so that the result broadcasts back. A matrix product is a table of dot products: an $m \times k$ matrix times a $k \times n$ matrix gives an $m \times n$ matrix at a cost of about $2mkn$ FLOPs. Norms measure the size of errors and weights. The ℓ_2 norm is the Euclidean length, and a dot product divided by the two lengths is the cosine of the angle between the vectors. Eigenvectors are the directions that a matrix only stretches, by a factor λ per multiplication, so repeated multiplication scales them by powers of λ, which is why deep networks must keep the stretch of their layers close to 1.

1.3.7 Exercises

1. **Transposes and products.** Prove the following statements.
    1. $(\mathbf{A}\mathbf{B})^\top = \mathbf{B}^\top\mathbf{A}^\top$ whenever the product is defined. Compare the entries of both sides using Equation 1.6.
    2. For every matrix $\mathbf{X}$, the matrix $\mathbf{X}\mathbf{X}^\top$ is symmetric. For the minibatch $\mathbf{X}$ of this section, state its shape and what its entry in row i and column j measures.
2. **Identity and inverse.** The scaling of Figure 1.9 has the matrix with rows $(2, 0)$ and $(0, 1/2)$, and the shear has the rows $(1, 1)$ and $(0, 1)$.
    1. Write down the inverse of each, check by hand that $\mathbf{A}^{-1}\mathbf{A} = \mathbf{I}$, and describe in words what each inverse does to the parallelogram in the figure.
    2. Show that the 2×2 matrix of all ones has no inverse: find a nonzero vector that it sends to $\mathbf{0}$, and explain why no matrix can undo that.
    3. The rotation by $90°$ has the columns $(0, 1)$ and $(-1, 0)$. Guess its inverse from the picture, and verify your guess.

3. code **Shape prediction.** Without running code, predict the result of each operation below, then check your answers. The tensor `images` has shape $(256, 1, 28, 28)$, and `X` has shape $(256, 784)$, as in this section.

    1. `len(images)`, and the shapes of the sums of `images` along axes 2 and 3 together (`axis=(2, 3)`), without and with `keepdims=True`. Which of the two results can divide `images` by broadcasting, and what does the division compute?
    2. `X - X.mean(axis=1)`. What goes wrong, and how does `keepdims` fix it?
    3. The same expression for a square minibatch of 784 images, with `X` of shape $(784, 784)$. Show that it runs without error, and write the entry in row i and column j of the result in terms of the entries of `X`. Why is this outcome worse than an error?

4. **A pairwise-obtuse triple.** The following questions ask how negative the pairwise dot products of several vectors can be.

    1. Determine whether three vectors $\mathbf{u}$, $\mathbf{v}$, $\mathbf{w}$ in the plane can satisfy $\mathbf{u}^\top \mathbf{v} < 0$, $\mathbf{v}^\top \mathbf{w} < 0$, and $\mathbf{u}^\top \mathbf{w} < 0$ simultaneously. Give a concrete numeric example or a short argument that none exists.
    2. Suppose $k \geq 2$ unit vectors $\mathbf{u}_1, \dots, \mathbf{u}_k$ have the same dot product c for every pair. Expand $\|\mathbf{u}_1 + \cdots + \mathbf{u}_k\|_2^2$ to show that $c \geq -1/(k-1)$. What does this bound say about the cosine similarity of ten class templates that are all equally far apart?

    Adapted from MIT 18.06, problem set 1, Section 1.2, Problem 28.

5. **Manhattan distance.** In midtown Manhattan, avenues and streets form a rectangular grid, and a pedestrian can walk only along them. Adjacent avenues lie about 250 m apart and adjacent streets about 80 m apart.

    1. Two intersections differ by Δa avenues and Δs streets. Write the length of the shortest walk between them and the straight-line distance between them as two norms of one displacement vector measured in meters.
    2. Prove that, for every $\mathbf{x} \in \mathbb{R}^n$,

    $$\|\mathbf{x}\|_2 \leq \|\mathbf{x}\|_1 \leq \sqrt{n}\, \|\mathbf{x}\|_2, \tag{1.16}$$

    and identify the vectors for which each bound holds with equality.

    3. Find the largest factor by which the shortest walk can exceed the straight-line distance, and the ratio $\Delta s / \Delta a$ at which this factor is attained.

6. code **Norms of higher-order tensors.** The vector norm functions of this section also accept tensors with three or more axes (in PyTorch, use `torch.linalg.vector_norm`; `matrix_norm` treats a tensor with three axes as a stack of matrices).

    1. Call the vector norm with its default arguments on a tensor with three axes. State in words what it computes, and confirm your statement by computing the same number directly from the tensor's entries.
    2. A model stores its parameters in several tensors of different shapes. Flatten each tensor and concatenate the results into one vector. Express the ℓ_2 norm of that vector in terms of the norms of the individual tensors, and confirm the formula numerically for tensors of shapes $(3, 4)$, $(4,)$, and $(2, 3, 5)$.

7. **Matrix-product order and cost.** Consider three large matrices $\mathbf{A} \in \mathbb{R}^{1000\times 50000}$, $\mathbf{B} \in \mathbb{R}^{50000\times 20}$, and $\mathbf{C} \in \mathbb{R}^{20\times 10000}$. You want the product $\mathbf{ABC}$. Count the FLOPs and the size of the intermediate matrix for $(\mathbf{AB})\mathbf{C}$ and for $\mathbf{A}(\mathbf{BC})$, and state by what factors the two orders differ in work and in memory.

8. code **Pairwise distances.** ☒ Two sets of 50 vectors in 8 dimensions are stored as the rows of $\mathbf{X}, \mathbf{Y} \in \mathbb{R}^{50\times 8}$. Compute the 50×50 squared Euclidean distances between every row of $\mathbf{X}$ and every row of $\mathbf{Y}$ in two ways, without loops.

    1. First, broadcast the differences of all pairs into a tensor of shape $(50, 50, 8)$ and sum over its last axis. Second, use $\|\mathbf{x}-\mathbf{y}\|^2 = \|\mathbf{x}\|^2 + \|\mathbf{y}\|^2 - 2\mathbf{x}^\top\mathbf{y}$, which needs one matrix product and no intermediate tensor of that shape. Confirm that the two results agree to floating-point tolerance.
    2. Count the numbers that each version stores along the way when each set holds 10,000 vectors, and decide which version you would use.
    3. Explain why the matrix-product version can return slightly negative values, and how to correct them.

    Adapted from Stanford CS231n, Assignment 1, k-nearest-neighbor distances, and fast.ai, Deep Learning from the Foundations, Lesson 8.

9. **Rotation matrices.** For an angle θ, consider the rotation matrix

$$\mathbf{R}_\theta = \begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix}. \tag{1.17}$$

    1. Show that $\mathbf{R}_\alpha\mathbf{R}_\beta = \mathbf{R}_{\alpha+\beta}$, and interpret this identity as one rotation applied after another.
    2. Show that $\|\mathbf{R}_\theta\mathbf{x}\|_2 = \|\mathbf{x}\|_2$ for every $\mathbf{x} \in \mathbb{R}^2$. State how the norm of $\mathbf{R}_\theta^k\mathbf{x}$ depends on k, and contrast this with repeated multiplication by the matrix $\mathbf{S}$ of this section.
    3. Show that $\mathbf{R}_\theta$ has no real eigenvalue unless θ is a multiple of 180°.

10. code **Power iteration.** ☒ Implement power iteration, which multiplies a random starting vector by a matrix and rescales the result to length 1 after every step, for the symmetric matrix $\mathbf{A}$ with rows $(1, 2, 3)$, $(2, 0, 4)$, and $(3, 4, 5)$.

    1. Report the first iteration at which the absolute value of the cosine similarity between successive iterates exceeds 0.999. Compare the final vector with the eigenvector of the largest eigenvalue returned by `eigh`, and explain why the two may differ in sign.
    2. Measure the factor by which the sine of the angle between the iterate and that eigenvector shrinks per step. Express it in terms of the eigenvalues of $\mathbf{A}$, and use it to explain the iteration count from the first part.
    3. Run the same iteration on $\mathbf{A} - 9\mathbf{I}$, where $\mathbf{I}$ is the identity matrix. Which eigenvalue and eigenvector does it find now? What happens to the sign of successive iterates? Explain both.

    Adapted from MIT 18.06, problem set 1, Section 2.1, Problems 29–30.

1.4 Calculus

Training a model means adjusting its parameters until the loss, a number that measures how badly the model fits its training data, is small. Suppose that a model has a single parameter, currently set to 1, and that its loss is 1. Should the parameter go up or down, and how much will the loss change if the parameter moves by 0.01? With one parameter we could try both directions and compare. A deep network has millions of parameters, and probing each of them separately costs one evaluation of the loss per parameter for every update, which is far too slow.

Calculus answers both questions with a formula instead of a probe. The derivative of the loss with respect to a parameter is the rate at which the loss changes when that parameter changes. Its sign tells us which way to move, and its size tells us how fast the loss responds. With many parameters, the derivatives form a vector, the gradient, which points in the direction in which the loss increases fastest. A small step against the gradient is the basic move of nearly every training algorithm in this book. Since a network computes its loss through many nested functions, we also need the chain rule, which assembles the derivative of a composition from the derivatives of its parts.

We develop these three tools on a running example small enough that every number in it can be checked by hand, using the vectors and dot products of Section 1.3. The next section lets the framework apply them for us. Note that calculus only tells us how to fit a model to its training data. Whether the fitted model predicts well on data that it has not seen, the problem of generalization, is a question for later chapters.

1.4.1 Derivatives

1.4.1.1 The Derivative as a Limit

Let us make the opening question concrete. A model predicts the same number w for every example, and we measure its error on three examples with labels 1, 1, and 0 by the sum of the squared differences between prediction and label:

$$L(w) = (w-1)^2 + (w-1)^2 + (w-0)^2 = 3w^2 - 4w + 2. \tag{1.18}$$

At $w = 1$ the first two predictions are correct and the third is off by one, so $L(1) = 1$.

```
def loss(w):
    return 3 * w ** 2 - 4 * w + 2
```

To see how the loss responds when w changes, we move w from 1 to $1 + h$ and divide the change in the loss by the change in the parameter. The ratio

$$\frac{L(1+h) - L(1)}{h} \tag{1.19}$$

is the average rate of change of L between 1 and $1 + h$: the number of units by which the loss changes per unit of w. We compute it for h from 10^{-1} down to 10^{-5}.

```
for h in 10.0 ** -np.arange(1, 6):
    print(f'h = {h:.0e}   ratio = {(loss(1 + h) - loss(1)) / h:.5f}')
```

```
h = 1e-01   ratio = 2.30000
h = 1e-02   ratio = 2.03000
h = 1e-03   ratio = 2.00300
h = 1e-04   ratio = 2.00030
h = 1e-05   ratio = 2.00003
```

The ratios approach 2, and algebra shows why. Expanding $L(1+h) = 3(1+h)^2 - 4(1+h) + 2 = 1 + 2h + 3h^2$ gives

$$\frac{L(1+h) - L(1)}{h} = 2 + 3h, \tag{1.20}$$

which is the printed sequence $2.3, 2.03, 2.003, \ldots$ and which comes as close to 2 as we like once h is small enough. The value 2 is the *derivative* of L at $w = 1$, written $L'(1) = 2$. In general, for a function $f : \mathbb{R} \to \mathbb{R}$ that maps a number to a number, the derivative of f at x is

$$f'(x) = \lim_{h \to 0} \frac{f(x+h) - f(x)}{h}. \tag{1.21}$$

The *limit* on the right is the value that the ratio approaches as h shrinks toward zero, from above and from below. The ratio itself is never evaluated at $h = 0$, where it would be $0/0$. When the limit exists, f is *differentiable* at x.

The ratio has a geometric meaning. It is the slope of the *secant*, the line through the two points $(x, f(x))$ and $(x+h, f(x+h))$ on the graph of f. As h shrinks, the second point slides along the graph toward the first, and the secant turns into the *tangent*, the line that touches the graph at $(x, f(x))$. The derivative is the slope of the tangent (Figure 1.14).

The device of the limit is much older than the derivative. Archimedes found the area of a circle of radius r by squeezing it between inscribed and circumscribed regular polygons with more and more sides (Figure 1.15 shows inscribed ones). An inscribed polygon with n sides splits into n triangles that meet at the center. As n grows, the height of each triangle approaches r and the bases add up to nearly the circumference $2\pi r$, so the area of the polygon approaches $\frac{1}{2} \cdot 2\pi r \cdot r = \pi r^2$. The derivative is found in the same way. We cannot compute a rate of change at a single point directly, so we compute the slopes of secants over shrinking intervals and take the limit.

Not every function is differentiable everywhere. The absolute value $|x|$ has no derivative at $x = 0$: the ratio $|h|/h$ equals 1 for every $h > 0$ and -1 for every $h < 0$, so it approaches no single value. Classification accuracy, the fraction of correctly classified examples, is worse. A small change of a parameter rarely changes any predicted class, so the accuracy stays constant and its derivative is zero wherever it exists. At the parameter values where a prediction flips, the accuracy jumps, and the derivative does not exist. Neither tells us which way to move. We therefore train with a differentiable *surrogate* loss, such as the squared error above, and report the accuracy separately.

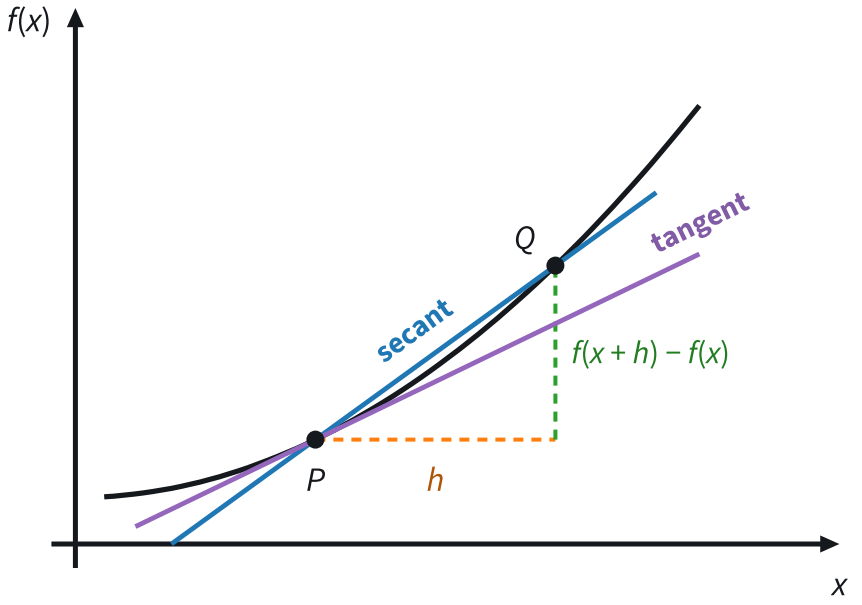


Figure 1.14: The difference quotient $(f(x+h)-f(x))/h$ is the slope of the secant through P and Q. As $h \to 0$, Q slides toward P and the secant turns into the tangent at P, whose slope is $f'(x)$.

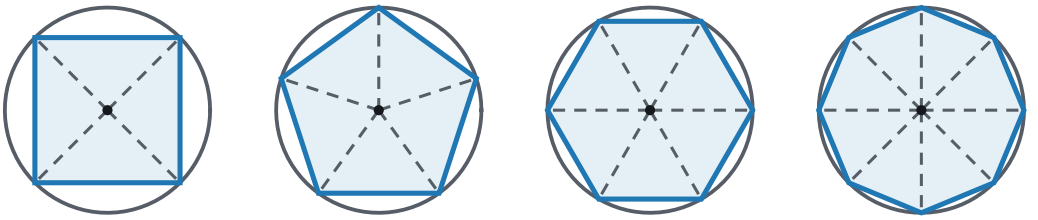


Figure 1.15: Regular polygons with 4, 5, 6, and 8 sides inscribed in a circle. As the number of sides grows, the area of the polygon approaches the area of the circle.

1.4.1.2 Rules of Differentiation

Computing limits is tedious. Luckily, we almost never need to: a few derivatives of basic functions, combined with a few rules, cover every function built from them. For $y = f(x)$ we write the derivative either as $f'(x)$ or as $\frac{dy}{dx}$. The second form, due to Leibniz, recalls the ratio of a small change in y to a small change in x, and it names the variable we differentiate with respect to, which helps once several variables are involved. The derivatives of the basic functions are

$$\frac{d}{dx}C = 0, \quad \frac{d}{dx}x^n = nx^{n-1}, \quad \frac{d}{dx}e^x = e^x, \quad \frac{d}{dx}\log x = \frac{1}{x},$$

for any constant C and any real exponent n, wherever the function is defined (the logarithm, for one, requires $x > 0$). A constant does not change, so its rate of change is zero, and e^x is the function that grows at a rate equal to its own value.

Multiples, sums, products, and quotients of differentiable functions f and g are differentiable, with

$$\begin{aligned}
\frac{d}{dx}[Cf(x)] &= Cf'(x) && \text{constant multiple rule,}\\
\frac{d}{dx}[f(x)+g(x)] &= f'(x)+g'(x) && \text{sum rule,}\\
\frac{d}{dx}[f(x)g(x)] &= f'(x)g(x)+f(x)g'(x) && \text{product rule,}\\
\frac{d}{dx}\frac{f(x)}{g(x)} &= \frac{f'(x)g(x)-f(x)g'(x)}{g(x)^2} && \text{quotient rule,}
\end{aligned}$$

where the quotient rule requires $g(x) \neq 0$. The difference rule, $(f-g)' = f' - g'$, follows from the sum rule and the constant multiple rule with $C = -1$. The sum rule says that rates of change add. The product rule is the only one that may look arbitrary, and Figure 1.16 shows why it is not: $f(x)g(x)$ is the area of a rectangle, and when both sides grow a little, the area gains two thin strips, one per factor, plus a corner too small to matter. Each rule follows from the limit definition; Section 34.1 gives the proofs and a longer table of derivatives.

For our loss the rules give

$$L'(w) = \frac{d}{dw}\left[3w^2 - 4w + 2\right] = 3 \cdot 2w - 4 \cdot 1 + 0 = 6w - 4, \tag{1.22}$$

and $L'(1) = 2$, the value that the ratios approached. Unlike a limit computed at one point, the formula gives the derivative at every w at once.

1.4.1.3 Linear Approximation and Minima

The definition Equation 1.21 says that for small h the ratio $(f(x+h) - f(x))/h$ is close to $f'(x)$. Multiplying by h turns this into a prediction of how f changes:

$$f(x+h) \approx f(x) + f'(x)\,h. \tag{1.23}$$

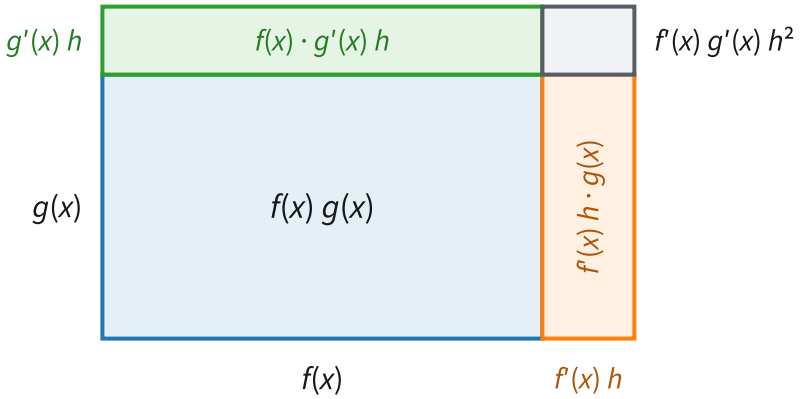


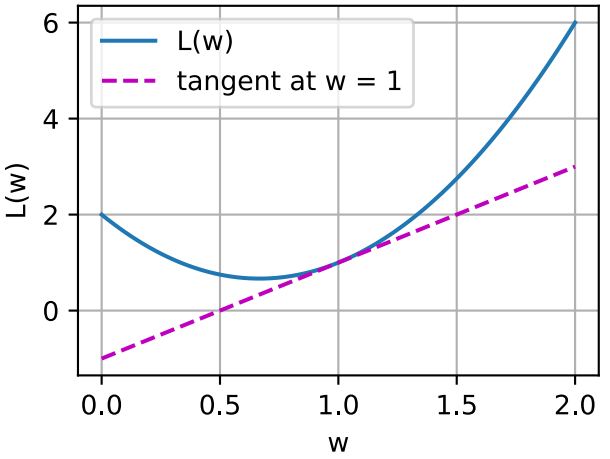
Figure 1.16: The product rule as the area of a rectangle. When x grows by h, the sides $f(x)$ and $g(x)$ grow by about $f'(x)h$ and $g'(x)h$, and the area grows by two thin strips and a corner of area $f'(x)g'(x)h^2$, which is negligible for small h.

The right-hand side is the tangent line at x, so near x the function stays close to its tangent. This approximation answers the opening question. At $w = 1$ the derivative $L'(1) = 2$ is positive, so increasing w by 0.01 raises the loss by about 0.02, and decreasing w by 0.01 lowers it by about as much. To lower the loss, we move w against the sign of the derivative. A step of 0.1 times the derivative, against its sign, moves w by $-0.1 \cdot L'(1) = -0.2$ to 0.8, where $L(0.8) = 0.72$. The approximation predicted $1 - 2 \cdot 0.2 = 0.6$. Where does the gap of 0.12 come from? It is the term $3h^2$ of $L(1 + h) = 1 + 2h + 3h^2$, with $h = -0.2$, which the tangent ignores. It grows with the square of the step, which is why steps must be small.

The derivative also locates the best parameter. At a minimum of a differentiable function, the derivative is zero, since otherwise a small step against its sign would lower the function further. Solving $L'(w) = 6w - 4 = 0$ gives $w = 2/3$, the mean of the three labels, with loss $L(2/3) = 2/3$. The converse does not hold in general: the derivative is also zero at a maximum and at points such as $x = 0$ for x^3. Our parabola, however, curves upward everywhere, so $w = 2/3$ is its minimum.

We plot L together with its tangent at $w = 1$, the line $L(1) + L'(1)(w - 1) = 2w - 1$, with `d2l.plot`, which draws several curves in one figure; its code is in the online appendix "Utility Functions and Classes".

```
w = np.linspace(0, 2, 101)
d2l.plot(w, [loss(w), 2 * w - 1], 'w', 'L(w)',
         legend=['L(w)', 'tangent at w = 1'])
```



Near $w = 1$ the parabola and its tangent are hard to tell apart. Farther away the curvature separates them, and the lowest point of the parabola lies at $w = 2/3$, where its own tangent would be horizontal.

1.4.2 Partial Derivatives and Gradients

A model with one parameter is rare. Let us therefore extend the derivative to functions of many variables. The extension turns out to answer a question that a single parameter could not even pose: among all the directions in which the parameters could move, which one lowers the loss fastest?

1.4.2.1 Partial Derivatives

Let us give our model a second parameter. The three examples now come with inputs 0, 1, and 2, and the model predicts $w_1 + w_2 \cdot \text{input}$, a line with intercept w_1 and slope w_2. The loss is again the sum of the squared errors,

$$L(w_1, w_2) = (w_1 - 1)^2 + (w_1 + w_2 - 1)^2 + (w_1 + 2w_2 - 0)^2.$$

Setting $w_2 = 0$ turns the line into the constant model, so $L(w_1, 0)$ is the loss $3w_1^2 - 4w_1 + 2$ from before. At $(w_1, w_2) = (1, 0)$ the loss is again 1, and the opening question returns in two parts: how does the loss change when we change w_1, and how when we change w_2?

The answer is the derivative with respect to one variable while all others are held fixed. For a function $f(x_1, \dots, x_n)$ of n variables, the *partial derivative* with respect to x_i is

$$\frac{\partial f}{\partial x_i} = \lim_{h \to 0} \frac{f(x_1, \dots, x_i + h, \dots, x_n) - f(x_1, \dots, x_i, \dots, x_n)}{h}. \tag{1.24}$$

Only the i^{th} variable moves. The curly ∂ is a reminder that f has other variables, which stay fixed. Since the other variables act as constants, a partial derivative is simply an ordinary derivative, and the rules of differentiation apply. Geometrically, fixing all variables but one cuts a curve out of the graph of f, and the partial derivative is the slope of that curve.

For the line model, holding $w_2 = 0$ leaves the curve $L(w_1, 0) = 3w_1^2 - 4w_1 + 2$, whose derivative at $w_1 = 1$ we know, so $\partial L/\partial w_1 = 2$. Holding $w_1 = 1$ leaves $L(1, w_2) = w_2^2 + (1 + 2w_2)^2 = 5w_2^2 + 4w_2 + 1$, with derivative $10w_2 + 4$, so $\partial L/\partial w_2 = 4$ at $w_2 = 0$. Increasing the slope raises the loss twice as fast as increasing the intercept (Figure 1.17).

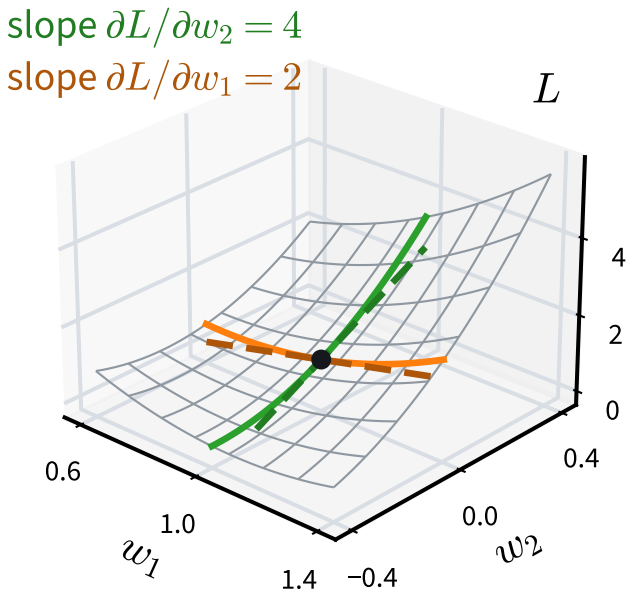


Figure 1.17: The loss $L(w_1, w_2)$ of the line model near $\mathbf{w} = (1, 0)$ (black dot). Holding $w_2 = 0$ cuts out the orange curve, the loss of the constant model; holding $w_1 = 1$ cuts out the green curve. Their tangents at the dot (dashed) have slopes $\partial L/\partial w_1 = 2$ and $\partial L/\partial w_2 = 4$.

We can check these numbers as we checked $L'(1)$: evaluate the ratio $(f(x + h) - f(x))/h$ for one small, fixed h, moving one parameter at a time. Such a ratio is called a *finite difference*. For the code we write the model with vectors and matrices. Each example becomes a row $(1, \text{input})$ of a matrix $\mathbf{X}$, the labels form a vector $\mathbf{y}$, and the parameters a vector $\mathbf{w} = (w_1, w_2)$:

$$\mathbf{X} = \begin{bmatrix} 1 & 0 \\ 1 & 1 \\ 1 & 2 \end{bmatrix}, \quad \mathbf{y} = \begin{bmatrix} 1 \\ 1 \\ 0 \end{bmatrix}, \quad L(\mathbf{w}) = \|\mathbf{X}\mathbf{w} - \mathbf{y}\|^2. \tag{1.25}$$

Entry i of the matrix–vector product $\mathbf{X}\mathbf{w}$ is the prediction $w_1 + w_2 \cdot \text{input}_i$, so $\mathbf{X}\mathbf{w} - \mathbf{y}$ is the vector of errors, and the squared norm adds up their squares. We use NumPy in float64: finite differences need every digit they can get, as we will see shortly.

```
X = np.array([[1.0, 0.0], [1.0, 1.0], [1.0, 2.0]])  # rows [1, input]
y = np.array([1.0, 1.0, 0.0])

def line_loss(w):
```

```
    return np.sum((X @ w - y) ** 2)

def numerical_gradient(f, w, h=1e-6):
    """Finite differences of f at w, moving one coordinate at a time."""
    grad = np.zeros_like(w)
    for i in range(len(w)):
        step = np.zeros_like(w)
        step[i] = h
        grad[i] = (f(w + step) - f(w)) / h
    return grad
```

```
w = np.array([1.0, 0.0])
numerical_gradient(line_loss, w)
```

```
array([2.000003, 4.000005])
```

The finite differences agree with 2 and 4 to within 10^{-5}. They also show what the opening called too slow: the loop evaluates the loss once more for every parameter. For a model with 10^8 parameters, one estimate of all the partial derivatives costs 10^8 evaluations of the loss, and training needs such an estimate at every step.

1.4.2.2 Precision of Finite Differences

Cost is not the only weakness of finite differences. Their error, $3h$ for the loss $L(w)$ of the constant model, shrinks with h, so why not simply choose a tiny h? In exact arithmetic this would work. On a computer it does not, and the reason is worth understanding, since it limits every numerical check of a gradient. A floating-point number keeps a fixed number of significant digits: about 16 decimal digits in the 64-bit format (`float64`, the default in NumPy) and about 7 in the 32-bit format (`float32`) that deep learning frameworks use by default (Section 1.1). The relative rounding error of each arithmetic operation is at most about ϵ, where $\epsilon \approx 2 \times 10^{-16}$ for float64 and $\epsilon \approx 10^{-7}$ for float32.

When h is small, $L(1 + h)$ and $L(1)$ agree in their leading digits, and subtracting them cancels those digits. The rounding errors of the two values survive the subtraction. Each is of order ϵ, since the loss is about 1 here, and dividing by the small h turns them into an error of up to about ϵ/h in the ratio. Shrinking h therefore reduces one error, the *truncation error* $3h$, and increases the other, the *rounding error* ϵ/h. We measure the total error $|(L(1 + h) - L(1))/h - 2|$ for h from 10^{-1} to 10^{-16}, first in float64 with NumPy and then in float32 with the framework's tensors.

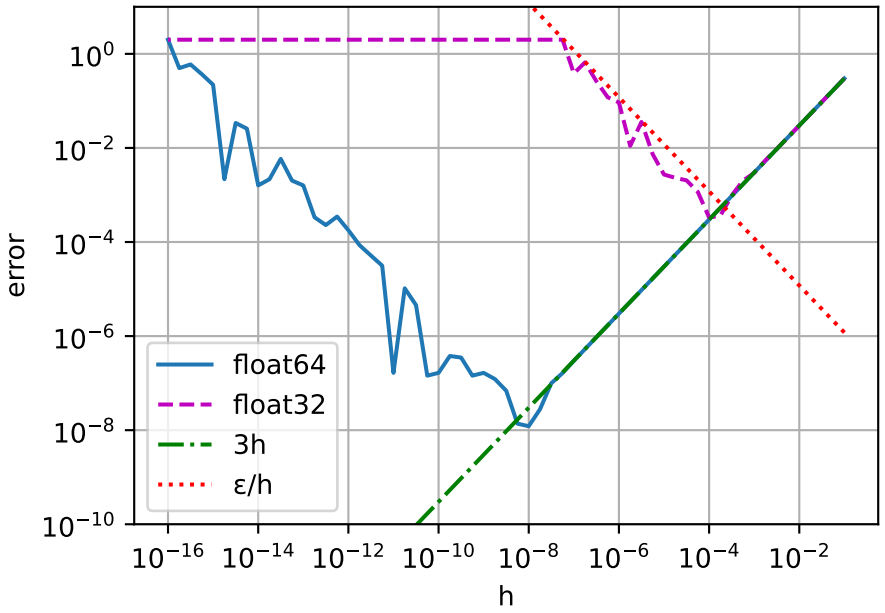
```
def quotient_error(f, x, df, h):
    """Error of (f(x + h) - f(x)) / h as an estimate of f'(x) = df."""
    return abs((f(x + h) - f(x)) / h - df)

h = np.logspace(-1, -16, 61)  # 61 values of h, evenly spaced on a log scale
```

```
h32 = torch.tensor(h, dtype=torch.float32)
```

We draw the two error curves with `d2l.plot`. Two reference lines show the truncation error $3h$ and the bound ϵ/h on the rounding error in float32.

```
eps32 = np.finfo(np.float32).eps
d2l.plot(h, [quotient_error(loss, 1, 2, h), quotient_error(loss, 1, 2, h32),
             3 * h, eps32 / h], 'h', 'error', xscale='log', yscale='log',
         ylim=(1e-10, 1e1), legend=['float64', 'float32', '3h', ' /h'],
         figsize=(5, 3.5))
```



Both axes are logarithmic, and the plot is easiest to read from the right. Where h is large, both curves follow the line $3h$ of slope 1: the truncation error does not depend on the precision. Each curve leaves this line where rounding takes over, and from there on a smaller h makes matters worse. The float32 error jumps from one h to the next but stays below the line ϵ/h of slope -1, and the float64 error does the same below the corresponding line for its much smaller ϵ, which is not drawn. Finally, once h is so small that $1 + h$ rounds to 1 itself, the computed ratio is 0 and the error is 2. In float32 this happens for every h below about 10^{-7}, which is why the left half of that curve is flat.

The best h balances the two errors, $3h \approx \epsilon/h$, which gives $h \approx \sqrt{\epsilon/3}$. In float64 this is near $h = 10^{-8}$, with an error of about 10^{-8}: half of the 16 available digits are lost. In float32 the best h is near 10^{-4}, and the derivative is accurate to only three or four digits.

Cost and precision are the two reasons why frameworks do not compute derivatives by finite differences. They apply instead the rules of differentiation to the operations that computed the loss, combined by the chain rule of Section 1.4.3, and obtain every digit that the arithmetic allows. Finite differences keep one job, which they do well: checking such a derivative, where three correct digits are plenty.

1.4.2.3 The Gradient

Each partial derivative tells us how the loss responds to one parameter while the others stay fixed. Training changes all the parameters at once, so it needs all of these numbers together.

Collecting the partial derivatives of f with respect to all n variables $\mathbf{x} = (x_1, \dots, x_n)$ gives a vector, the *gradient* of f:

$$\nabla_{\mathbf{x}} f(\mathbf{x}) = \left(\frac{\partial f}{\partial x_1}, \frac{\partial f}{\partial x_2}, \dots, \frac{\partial f}{\partial x_n} \right). \tag{1.26}$$

We write $\nabla f(\mathbf{x})$ when the variable is clear. The gradient has one entry per variable, so it has the same shape as $\mathbf{x}$; the gradient of a loss has the shape of the parameters. For the line model, $\nabla L(\mathbf{w}) = (2, 4)$ at $\mathbf{w} = (1, 0)$.

The gradient extends the approximation Equation 1.23 to moves of all variables at once. Suppose that we change $\mathbf{x}$ by a small vector $\mathbf{h}$, one coordinate after another. Changing x_1 by h_1 changes f by about $h_1 \, \partial f / \partial x_1$, then changing x_2 by h_2 adds about $h_2 \, \partial f / \partial x_2$, and so on. The changes add up to a dot product:

$$f(\mathbf{x} + \mathbf{h}) \approx f(\mathbf{x}) + \sum_{i=1}^{n} \frac{\partial f}{\partial x_i} h_i = f(\mathbf{x}) + \nabla f(\mathbf{x})^\top \mathbf{h}. \tag{1.27}$$

The function changes by the dot product of the gradient with the move. Our argument is not quite a proof. It evaluates each partial derivative at a point that has already moved a little, so the approximation needs partial derivatives that change continuously; Section 34.2 states the condition and proves the approximation. For the line model, lowering both parameters by 0.01 predicts a loss of $1 + 2 \cdot (-0.01) + 4 \cdot (-0.01) = 0.94$, and the exact value is $L(0.99, -0.01) = 0.9414$.

1.4.2.4 Steepest Descent

Among all moves of a given small length, which one lowers f the most? We write the move as $\mathbf{h} = t\mathbf{d}$, where the unit vector $\mathbf{d}$ gives the direction and the small number $t > 0$ gives the length. By Equation 1.27, f changes by about $t \, \nabla f(\mathbf{x})^\top \mathbf{d}$. Recall from Equation 1.7 that the dot product of two vectors equals the product of their lengths and the cosine of the angle between them, $\mathbf{a}^\top \mathbf{b} = \|\mathbf{a}\| \|\mathbf{b}\| \cos\theta$. Since $\|\mathbf{d}\| = 1$,

$$f(\mathbf{x} + t\mathbf{d}) - f(\mathbf{x}) \approx t \, \|\nabla f(\mathbf{x})\| \cos\theta,$$

where θ is the angle between $\mathbf{d}$ and the gradient. The change depends on the direction only through $\cos\theta$, and three directions stand out. The change is largest for $\theta = 0$, when $\mathbf{d}$ points along the gradient. The gradient is therefore the direction of *steepest ascent*, and its length $\|\nabla f(\mathbf{x})\|$ is the rate at which f grows in that direction. The change is most negative for $\theta = 180^\circ$, so the negative gradient $-\nabla f(\mathbf{x})$ is the direction of *steepest descent*.

The change is zero for $\theta = 90^\circ$: to first order, moving perpendicular to the gradient leaves f unchanged. The curves along which f is constant are called its *level sets*, like the contour lines of a map, so the gradient is perpendicular to the level sets and points uphill (Figure 1.18).

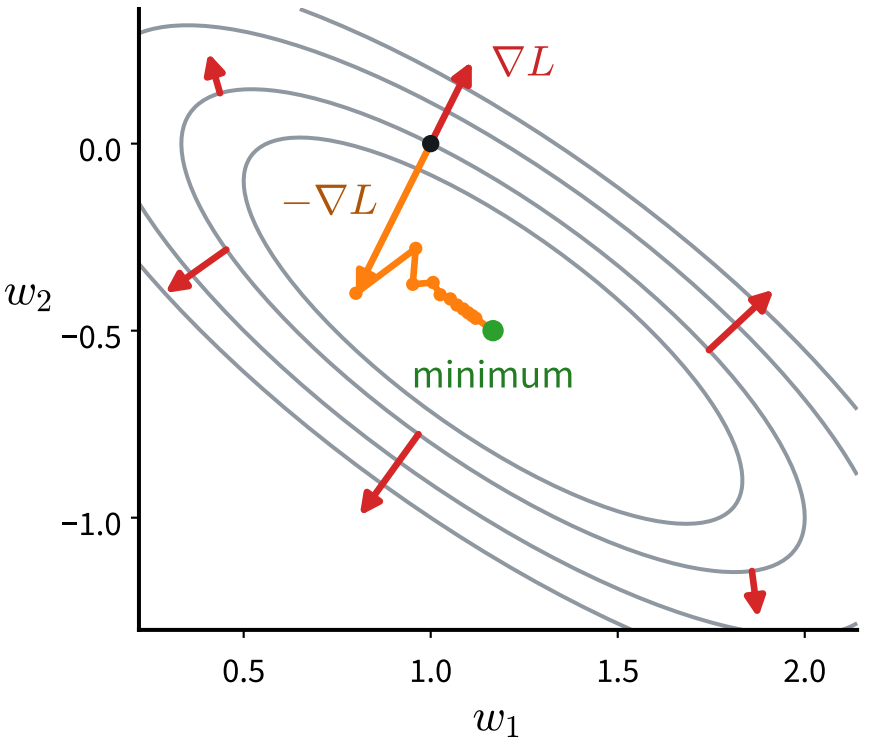


Figure 1.18: Level sets of the loss $L(w_1, w_2)$ of the line model. The gradients (red) are perpendicular to the level set through their base points and point uphill. From $\mathbf{w} = (1, 0)$ (black dot), gradient descent with $\eta = 0.1$ (orange) starts along $-\nabla L$, which does not point at the minimum, and zigzags toward it.

Gradient descent repeats a small step along the negative gradient,

$$\mathbf{w} \leftarrow \mathbf{w} - \eta \nabla L(\mathbf{w}), \tag{1.28}$$

where the *learning rate* $\eta > 0$ sets the size of the step; the factor 0.1 in the one-parameter step above was a learning rate. From $\mathbf{w} = (1, 0)$ with $\eta = 0.1$, one step reaches $(0.8, -0.4)$ and lowers the loss from 1 to 0.4. Note, however, that "steepest" is a local statement, and it says nothing about where the minimum lies. The minimum of the line model's loss lies at $(7/6, -1/2)$, and the negative gradient at $(1, 0)$ points 45° away from it, because the level sets are elongated ellipses rather than circles. Repeated steps still approach the minimum, but along a zigzag path. Later chapters refine the step (Chapter 9), but every refinement starts from the gradient.

1.4.2.5 Gradients of Common Functions

So far we obtained the gradient $(2, 4)$ of the line model one coordinate at a time. To find it at every $\mathbf{w}$ at once, without writing out three squares, we need the gradients of a linear and of a quadratic function, which recur throughout the book. Each follows from writing the function out coordinate by coordinate and differentiating, much as the scalar rules follow from the limit. For a constant vector $\mathbf{a}$, the linear function $\mathbf{a}^\top \mathbf{x} = \sum_i a_i x_i$ has partial derivative a_i with respect to x_i, so

$$\nabla_{\mathbf{x}} \, \mathbf{a}^\top \mathbf{x} = \mathbf{a}. \tag{1.29}$$

For a constant square matrix $\mathbf{A}$, the quadratic function $\mathbf{x}^\top \mathbf{A} \mathbf{x} = \sum_{i,j} x_i a_{ij} x_j$ contains x_k in the terms with $i = k$ and in those with $j = k$. Differentiating with respect to x_k gives $\sum_j a_{kj} x_j + \sum_i a_{ik} x_i$, the k^{th} entry of $\mathbf{A}\mathbf{x} + \mathbf{A}^\top \mathbf{x}$, so

$$\nabla_{\mathbf{x}} \, \mathbf{x}^\top \mathbf{A} \mathbf{x} = (\mathbf{A} + \mathbf{A}^\top)\mathbf{x}. \tag{1.30}$$

For a symmetric matrix this is $2\mathbf{A}\mathbf{x}$. The simplest case is the squared norm $\|\mathbf{x}\|^2 = \sum_i x_i^2$, with $\nabla_{\mathbf{x}} \|\mathbf{x}\|^2 = 2\mathbf{x}$. These are the vector versions of $\frac{d}{dx} ax = a$ and $\frac{d}{dx} ax^2 = 2ax$, and they are worth remembering; Section 34.4 derives many more such identities.

Now for the line model. Expanding the square, $L(\mathbf{w}) = \mathbf{w}^\top \mathbf{X}^\top \mathbf{X} \mathbf{w} - 2(\mathbf{X}^\top \mathbf{y})^\top \mathbf{w} + \mathbf{y}^\top \mathbf{y}$, and the symmetric matrix $\mathbf{X}^\top \mathbf{X}$ yields

$$\nabla L(\mathbf{w}) = 2\mathbf{X}^\top \mathbf{X} \mathbf{w} - 2\mathbf{X}^\top \mathbf{y} = 2\mathbf{X}^\top (\mathbf{X}\mathbf{w} - \mathbf{y}). \tag{1.31}$$

At $\mathbf{w} = (1, 0)$ the errors are $\mathbf{X}\mathbf{w} - \mathbf{y} = (0, 0, 1)$, and the gradient is $2 \cdot (1, 2) = (2, 4)$, as before.

1.4.3 The Chain Rule

A network computes its loss in stages. Each layer transforms the output of the previous one, and the loss compares the final output with the label. The loss is thus a *composition* of functions, and expanding it into one formula, as we just did for the line model, quickly becomes unmanageable. The *chain rule* differentiates a composition stage by stage instead.

1.4.3.1 Functions of One Variable

Suppose that $z = f(u)$ and $u = g(x)$ are differentiable, so that $z = f(g(x))$. Then

$$\frac{dz}{dx} = \frac{dz}{du}\frac{du}{dx}.$$

Rates of change multiply. If u changes twice as fast as x, and z changes three times as fast as u, then z changes six times as fast as x. In terms of the linear approximation Equation 1.23, a small change h in x changes u by about $g'(x)h$, which in turn changes z by about $f'(u)g'(x)h$. The proof in Section 34.1 makes this argument precise.

For example, the last term of the line model's loss, $(w_1+2w_2)^2$, is $z = u^2$ with $u = w_1+2w_2$. With respect to w_2, we have $dz/du = 2u$ and $\partial u/\partial w_2 = 2$, the input of the third example, so the term changes at the rate $2u \cdot 2 = 4$ at $(1, 0)$, where $u = 1$. The other two terms have zero error at that point and change at rate zero, which is why $\partial L/\partial w_2 = 4$.

1.4.3.2 Functions of Many Variables

In a network, a parameter usually affects the loss along many paths. Suppose that z depends on m intermediate variables $u_1, \dots, u_m$, each of which depends on n inputs $x_1, \dots, x_n$. A small change in x_i changes every u_j, and each of those changes reaches z. The chain rule adds up the contributions of all paths:

$$\frac{\partial z}{\partial x_i} = \sum_{j=1}^{m} \frac{\partial z}{\partial u_j}\frac{\partial u_j}{\partial x_i}. \tag{1.32}$$

Each term multiplies the rates along one path $x_i \to u_j \to z$, as in the one-variable chain rule. To write the sum in vector form, we collect the derivatives of the intermediate variables in an $m \times n$ matrix $\mathbf{J}$, the *Jacobian* of $\mathbf{u}$ with respect to $\mathbf{x}$, with entries $J_{ji} = \partial u_j/\partial x_i$. Row j of the Jacobian is the transposed gradient of u_j. The sum over j is then entry i of the matrix–vector product $\mathbf{J}^\top \nabla_{\mathbf{u}} z$:

$$\nabla_{\mathbf{x}} z = \mathbf{J}^\top \nabla_{\mathbf{u}} z. \tag{1.33}$$

The gradient with respect to the inputs is the gradient with respect to the intermediate variables, multiplied by the transposed Jacobian. For a linear map $\mathbf{u} = \mathbf{A}\mathbf{x}$, entry $u_j = \sum_i a_{ji}x_i$ has $\partial u_j/\partial x_i = a_{ji}$, so the Jacobian is $\mathbf{A}$ itself and $\nabla_{\mathbf{x}} z = \mathbf{A}^\top \nabla_{\mathbf{u}} z$.

The line model has this structure. Its loss $L = \|\mathbf{r}\|^2$ is a function of the vector of errors $\mathbf{r} = \mathbf{X}\mathbf{w} - \mathbf{y}$, which depends linearly on $\mathbf{w}$, with Jacobian $\mathbf{X}$. Since $\nabla_{\mathbf{r}} L = 2\mathbf{r}$, the chain rule gives

$$\nabla_{\mathbf{w}} L = \mathbf{X}^\top (2\mathbf{r}) = 2\mathbf{X}^\top (\mathbf{X}\mathbf{w} - \mathbf{y}), \tag{1.34}$$

the gradient Equation 1.31, obtained this time without expanding the square. The computation splits into two passes. The *forward pass* computes the errors and then the loss. The *backward pass* starts at the loss, computes its gradient with respect to the errors, and multiplies by the transposed Jacobian to reach the parameters.

```
r = X @ w - y    # forward: errors, from which the loss is r @ r
grad_r = 2 * r   # backward: gradient of the loss with respect to r
X.T @ grad_r     # backward: gradient with respect to w
```

```
array([2., 4.])
```

The result is $(2, 4)$ once more. The forward and the backward pass each cost one matrix–vector product, so the whole gradient of the line model costs about as much as a second evaluation of the loss, where finite differences need one evaluation per parameter. A deep network repeats this pattern layer by layer. The forward pass evaluates the layers from the input to the loss, and the backward pass multiplies by one transposed Jacobian per layer, from the loss back to the parameters. This backward application of the chain rule is called *backpropagation*.

What remains is the labor of deriving the Jacobian of every layer by hand, which is slow and error-prone for networks with dozens of layers and millions of parameters. It is also entirely mechanical: nothing in the backward pass above required insight, only care. Section 1.5 hands this labor to the framework, which records the forward pass and runs the backward pass for us, and explains why the gradient of a deep network, like that of the line model, costs only a small multiple of computing its loss.

1.4.4 Summary

Training needs to know how the loss changes when the parameters change, and for one parameter the derivative answers this. It is the limit of the ratio of the change in the loss to a small change in the parameter, which is the slope of the tangent, and for small steps $f(x+h) \approx f(x)+f'(x)h$. For many parameters, the partial derivatives form the gradient, whose negative is the direction of steepest descent, and gradient descent takes small steps along it. Estimating the gradient by finite differences costs one loss evaluation per parameter, and the ratio with a small h loses about half of the available digits in floating point, which in float32 leaves three or four. The chain rule differentiates compositions by multiplying rates along each path and adding over paths. In vector form it multiplies a gradient by transposed Jacobians, from the loss back to the parameters, which is backpropagation.

1.4.5 Exercises

1. **Derivatives from first principles.** The derivatives and rules of this section were stated without proof. Prove the following from the limit definition Equation 1.21.
    1. $f(x) = C$ for a constant C, and $f(x) = x^n$ for a positive integer n.
    2. $f(x) = e^x$ and $f(x) = \log x$, taking as given that $(e^h - 1)/h \to 1$ as $h \to 0$. For the logarithm, write $\log(x+h) - \log x = \log(1+h/x)$ and substitute $1+h/x = e^t$.
    3. The sum, product, and quotient rules for differentiable f and g, assuming $g(x) \neq 0$ for the quotient rule.
2. **Predicting with the tangent.** The square root has derivative $\frac{d}{dx}\sqrt{x} = 1/(2\sqrt{x})$, and $\sqrt{4} = 2$.
    1. Use Equation 1.23 at $x = 4$ to predict $\sqrt{4.1}$ and $\sqrt{4.5}$ without a calculator, then compare with the true values.

2. Both predictions err in the same direction. Explain why from the shape of the graph, and check that the error grows about as the square of the step, as it did for the loss in this section.

3. **Derivative of x^x.** For $x > 0$, the base and the exponent of $f(x) = x^x$ both vary, so neither the power rule nor the rule for an exponential applies.
   1. Explain why the power-rule answer $x \cdot x^{x-1}$ and the exponential-rule answer $x^x \log x$ are both wrong.
   2. Write $x^x = e^{x \log x}$ and compute $f'(x)$ with the chain rule. Show that the correct derivative is the sum of the two wrong answers, and explain why.

4. **Mean and median.** A model predicts the same number w for every example, and the labels are $y_1, \dots, y_n$.
   1. Show that the squared error $\sum_i (w - y_i)^2$ is minimized by the mean of the labels.
   2. Show that, wherever it exists, the derivative of the absolute error $\sum_i |w - y_i|$ equals the number of labels below w minus the number of labels above w. Where does it fail to exist? Conclude that a median of the labels minimizes the absolute error.
   3. For the labels 1, 1, 0, 10, find both minimizers. Which of them changes when the label 10 becomes 100?

5. **Gradient of the ℓ_2 norm.** Find the gradient of the function $f(\mathbf{x}) = \|\mathbf{x}\|$ for $\mathbf{x} \neq \mathbf{0}$ and show that it has length 1. State what happens at $\mathbf{x} = \mathbf{0}$ and why.

6. **Multivariate chain rule.** Let $z = u_1 u_2 u_3$, where $u_1 = x_1 + x_2$, $u_2 = x_1 - x_2$, and $u_3 = x_1 x_2$.
   1. Compute $\nabla_{\mathbf{u}} z$ and the Jacobian $\mathbf{J}$ of $\mathbf{u}$ with respect to $\mathbf{x}$, and obtain $\nabla_{\mathbf{x}} z$ as the product $\mathbf{J}^\top \nabla_{\mathbf{u}} z$.
   2. Substitute u_1, u_2, and u_3 into z, differentiate the result with respect to x_1 and x_2 directly, and compare with the product.

7. **Gradient-descent step.** Consider $f(\mathbf{x}) = \|\mathbf{x}\|^2$ and the gradient-descent step with learning rate $\eta > 0$,
$$\mathbf{x} \leftarrow \mathbf{x} - \eta \nabla f(\mathbf{x}). \tag{1.35}$$
   1. Starting from $\mathbf{x} = (1, 1)$, take one step for each of $\eta = 0.1$, $\eta = 1$, and $\eta = 2$, and compute f before and after the step.
   2. Write the step for this f in closed form for arbitrary $\mathbf{x}$ and η. Find the range of η for which a step decreases f at every $\mathbf{x} \neq \mathbf{0}$, and the value of η that reaches the minimizer in one step.
   3. Characterize the points at which a step leaves $\mathbf{x}$ unchanged, for this f and for a general differentiable f.

8. code **Central differences.** ⊠ The forward difference $(f(x+h) - f(x))/h$ of this section has a truncation error proportional to h. The central difference $(f(x+h) - f(x-h))/(2h)$ evaluates f on both sides of x. Write a central-difference counterpart of `quotient_error` and use the values of h from this section, in float64.
   1. For the loss $L(w) = 3w^2 - 4w + 2$ at $w = 1$, plot the errors of the forward and the central difference in one figure, and explain the shape of the new curve.
   2. Repeat both computations for $f(x) = e^x$ at $x = 0$, where $f'(0) = 1$. Compare the slopes of the large-h branches of the two curves and the h at which

each difference is most accurate, and explain both with the expansion $e^h = 1 + h + h^2/2 + h^3/6 + \cdots$.

9. code **Gradient checking.** ▣ Finite differences are too slow for training, but they are the standard test for a gradient derived or coded by hand.
   1. Write a function that estimates the gradient of $f : \mathbb{R}^n \to \mathbb{R}$ with central differences. Use it in float64 to check $\nabla_{\mathbf{x}} \mathbf{x}^\top \mathbf{A}\mathbf{x} = (\mathbf{A} + \mathbf{A}^\top)\mathbf{x}$ for a random 5×5 matrix $\mathbf{A}$ that is not symmetric, at a random $\mathbf{x}$, and report the largest relative error over the five entries.
   2. Check the incorrect formula $2\mathbf{A}\mathbf{x}$ in the same way. How large is the relative error now?
   3. Repeat the check of the correct formula in float32 with the framework's tensors, for $h = 10^{-1}, 10^{-2}, \dots, 10^{-6}$. Which h gives the smallest error, and why is it not the best h of this section's experiment? Which tolerance on the relative error separates the correct formula from the incorrect one in float32?

1.5 Automatic Differentiation

In Section 1.4 we computed the gradient of a loss by hand, and it took some care even for a model with two parameters. A deep network has millions of parameters spread over dozens of layers, and every change to its design would mean deriving and coding its gradient again. Can the computer do this work for us, given nothing more than the code that computes the loss?

Recall the line model of Section 1.4, whose loss is the squared norm of the errors $\mathbf{r} = \mathbf{X}\mathbf{w} - \mathbf{y}$ on three examples. At $\mathbf{w} = (1, 0)$ we found the gradient $\nabla_{\mathbf{w}} L = \mathbf{X}^\top (2\mathbf{r}) = (2, 4)$, but only after deriving a Jacobian and coding the product with it ourselves. A deep network would ask the same of us for every layer.

Luckily, deep learning frameworks do this work for us, and the idea behind it is simple. While our code computes the loss, the framework records each operation that it executes, such as a matrix–vector product or a sum, along with the inputs of the operation. The derivative of every such operation is built into the framework. Once the loss is known, the framework walks through the record backward, from the loss to the parameters, and applies the chain rule at each operation, as we did by hand for the line model. This procedure is called *automatic differentiation*, or *autograd* for short. It returns the derivative of the computation up to floating-point rounding, at a small multiple of the cost of computing the loss, whatever the number of parameters.

We first let the framework compute the gradient of the line model and compare it with $(2, 4)$. We then control which operations are recorded and differentiate programs with loops and branches. The end of the section goes further than the next chapters need: it explains why frameworks walk the record backward rather than forward.

```
import torch
```

1.5.1 Computing Gradients

1.5.1.1 The Forward Pass

We set up the line model with the framework's tensors. Each row $(1, \text{input})$ of $\mathbf{X}$ describes one of the three examples, $\mathbf{y}$ holds their labels, and the parameters start at $\mathbf{w} = (1, 0)$. The function `loss` computes $L(\mathbf{w}) = \|\mathbf{X}\mathbf{w} - \mathbf{y}\|^2$. It is the forward pass only: no derivative appears anywhere in the code.

```
X = torch.tensor([[1.0, 0.0], [1.0, 1.0], [1.0, 2.0]])  # [1, input]
y = torch.tensor([1.0, 1.0, 0.0])
w = torch.tensor([1.0, 0.0])

def loss(w):
    return ((X @ w - y) ** 2).sum()
```

Recording costs memory and time, so the framework records only the operations whose results depend on tensors that we mark as needing a gradient. We mark $\mathbf{w}$.

`requires_grad_(True)` marks `w`; the trailing underscore is PyTorch's convention for a method that changes its tensor in place. From now on every operation on `w` is recorded, and the backward pass will store the gradient in the attribute `w.grad`.

```
w.requires_grad_(True)
```

```
tensor([1., 0.], requires_grad=True)
```

We now run the forward pass.

```
L = loss(w)
L
```

```
tensor(1., grad_fn=<SumBackward0>)
```

The loss carries `grad_fn=<SumBackward0>`: it remembers the operation that produced it, the last one in the record.

The record forms a *computational graph* (Figure 1.19). Its nodes are the values computed along the way, from the parameters to the loss, and an edge runs from each input of an operation to its result. The framework records operations at the level at which it knows their derivatives, so the loss of the line model becomes four operations: a matrix–vector product, the subtraction of the labels, an elementwise square, and a sum.

1.5.1.2 The Backward Pass

We now ask the framework for the gradient of the loss at $\mathbf{w}$ and compare it with the $(2, 4)$ that we derived by hand.

Calling `backward` on the loss runs the backward pass and stores the gradient in `w.grad`.

```
L.backward()
w.grad
```

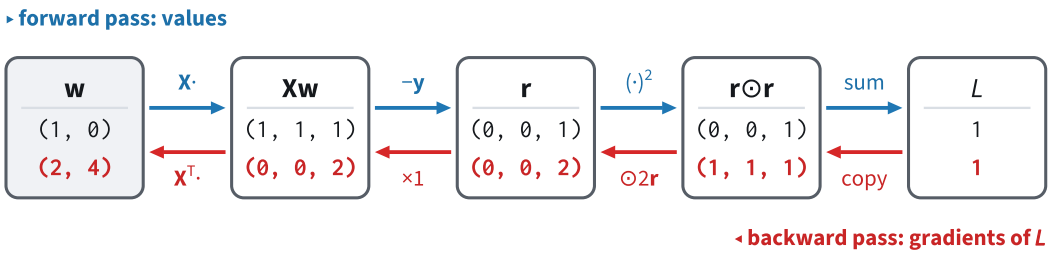


Figure 1.19: The computational graph of $L = \|\mathbf{X}\mathbf{w} - \mathbf{y}\|^2$ at $\mathbf{w} = (1, 0)$, with one node for $\mathbf{w}$ and for each value computed from it; the constants $\mathbf{X}$ and $\mathbf{y}$ are not drawn, and $\mathbf{r} \odot \mathbf{r}$ is the elementwise square. Each node shows its value (black) and the gradient of L with respect to it (red). The forward pass (blue) runs from left to right; the backward pass (red) starts from 1 at L and applies the local rule of each operation, from right to left.

```
tensor([2., 4.])
```

The result is $(2, 4)$, as it should be. How did the framework arrive at it without being told the formula $2\mathbf{X}^\top(\mathbf{X}\mathbf{w} - \mathbf{y})$? By hand we multiplied by two factors, $2\mathbf{r}$ and $\mathbf{X}^\top$. The backward pass of Figure 1.19 produces them from four local rules, one per operation. It starts from the derivative 1 of L with respect to itself. The sum copies this gradient to each of its three inputs, the square multiplies it entry by entry by $2\mathbf{r}$, subtracting the constant labels leaves it unchanged, and the matrix–vector product multiplies it by $\mathbf{X}^\top$. Each rule uses only the local derivative of one operation and values saved during the forward pass, here $\mathbf{r}$ and $\mathbf{X}$. This is why training needs more memory than prediction: the framework must keep such values until the backward pass has used them.

1.5.1.3 Gradient Accumulation

Training computes a new gradient at every step, so it matters what happens to the gradient of the previous step.

To find out, we run the forward and the backward pass a second time, at the same $\mathbf{w}$.

```
L = loss(w)
L.backward()
w.grad  # the new gradient was added to the old one
```

```
tensor([4., 8.])
```

The answer is $(4, 8)$, twice the gradient: PyTorch added the new gradient to the one already stored in `w.grad`. Accumulation is useful when a loss is a sum of parts whose gradients are computed separately, since calling `backward` on each part adds up their gradients. A training step that needs a fresh gradient must therefore first reset `w.grad`, for instance with `w.grad.zero_()`. A forgotten reset is a common bug, and no error message reveals it.

1.5.1.4 Vector-Valued Functions

So far the function that we differentiated returned a scalar. Often it returns a vector, for instance the loss of each example in a minibatch, before the losses are added up. What should the backward pass return then? It can differentiate only one number, so we have

to say which. The choice that frameworks offer is a weighted sum $\mathbf{v}^\top \mathbf{u} = \sum_j v_j u_j$ of the outputs $\mathbf{u}$, with a vector of weights $\mathbf{v}$ that we supply. With $\mathbf{v} = \mathbf{1}$, the vector of all ones, this is the plain sum of the outputs, and for per-example losses that is what we want: the gradient of the total loss. Let us try it on the per-example squared errors r_i^2 of the line model, whose sum is L.

PyTorch does not choose $\mathbf{v}$ for us: calling `backward` on a tensor that is not a scalar raises an error unless we pass $\mathbf{v}$ as the argument `gradient`. The name reflects the chain rule, since $\mathbf{v}$ is the gradient arriving from the rest of a larger computation. Passing a vector of ones is the same as calling `losses.sum().backward()`.

```
w.grad.zero_()
losses = (X @ w - y) ** 2  # one loss per example
losses.backward(gradient=torch.ones(3))  # = losses.sum().backward()
w.grad
```

```
tensor([2., 4.])
```

The result is $(2, 4)$, the gradient of the total loss.

It is instructive to write this in the terms of Section 1.4. Let $\mathbf{u}$ consist of m values computed from n inputs $\mathbf{x}$. Its derivative is the $m \times n$ Jacobian $\mathbf{J}$ with entries $J_{ji} = \partial u_j / \partial x_i$, whose row j is the transposed gradient of u_j. The backward pass started from $\mathbf{v}$ returns

$$\mathbf{J}^\top \mathbf{v} = \nabla_{\mathbf{x}} \left(\mathbf{v}^\top \mathbf{u} \right) = \sum_{j=1}^{m} v_j \nabla_{\mathbf{x}} u_j, \tag{1.36}$$

and it does so without ever computing $\mathbf{J}$ itself. This is the chain rule $\nabla_{\mathbf{x}} z = \mathbf{J}^\top \nabla_{\mathbf{u}} z$ for a scalar z computed from $\mathbf{u}$ whose gradient $\nabla_{\mathbf{u}} z$ is $\mathbf{v}$. Choosing $\mathbf{v} = \mathbf{e}_j$, the coordinate vector with a 1 in position j and 0 elsewhere (Section 1.3), makes the backward pass return the gradient of u_j alone, which is row j of the Jacobian. Section 1.5.4 uses this to compute whole Jacobians.

1.5.2 Controlling the Graph

Frameworks record every operation on a marked tensor, and sometimes that is more than we want. A value computed from the parameters may need to act as a constant, and some computations, such as the update of the parameters, should not be recorded at all.

1.5.2.1 Detaching Values

To *detach* a value is to keep the value but cut its connection to the parameters, so that the backward pass treats it as a constant. A small example shows what this does to a gradient. Let $u = x^2$ and $z = ux$ for each entry x of $\mathbf{x} = (0, 1, 2, 3)$, so that $z = x^3$. In the graph of Figure 1.20(a), x reaches z along two paths, through u and directly. The chain rule adds their contributions, $x \cdot 2x$ and u, to give $\partial z / \partial x = 3x^2$. If we replace u by a detached copy c, which has the same value but no history, the path through u disappears (Figure 1.20(b)), and $\partial z / \partial x = c = x^2$.

`u.detach()` returns a new tensor with the values of `u` and no history.

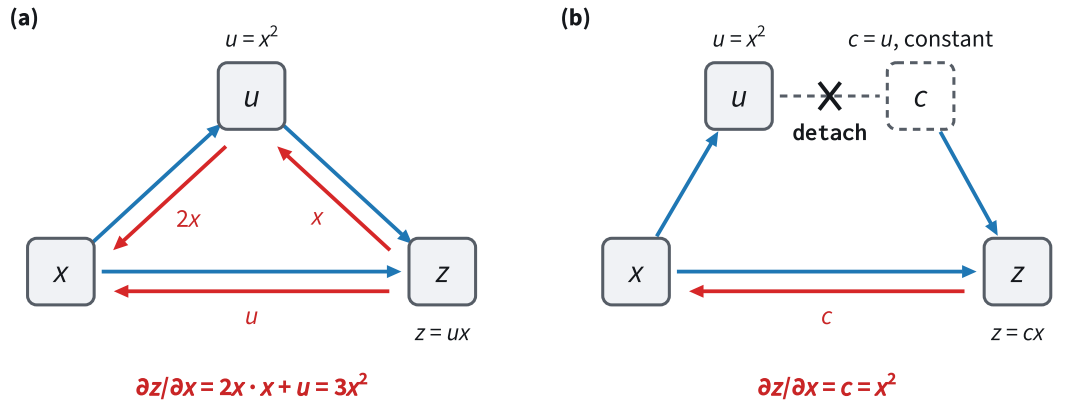


Figure 1.20: The product $z = ux$ with $u = x^2$, (a) attached and (b) detached. Blue arrows are operations of the forward pass; red arrows show the backward pass, labeled with the local derivatives it multiplies by. Detaching u creates a constant c with the same value and no connection to x.

```
x = torch.arange(4.0, requires_grad=True)
u = x * x
c = u.detach()  # values of u, no history
z = c * x
z.sum().backward()
x.grad
```

```
tensor([0., 1., 4., 9.])
```

The gradient is $(0, 1, 4, 9)$, which is $c = x^2$. Without the detach, both paths carry gradient.

Detaching did not change `u`, which keeps its history, so we can reuse it.

```
x.grad.zero_()
z = u * x
z.sum().backward()
x.grad
```

```
tensor([ 0.,  3., 12., 27.])
```

The gradient is now $3x^2 = (0, 3, 12, 27)$, although z has the same value $(0, 1, 8, 27)$ in both computations. Detaching changes the graph, not the values.

The typical use is a target that the model computes itself. Q-learning (Section 14.3), for instance, moves a prediction toward a target built from the model's own predictions. The target must then act as a constant, or else an update could lower the loss by moving the target toward the prediction, which improves nothing.

1.5.2.2 Disabling Gradient Tracking

When we need only values, as when a trained model makes predictions, recording wastes time and memory, and we switch it off. Updating the parameters is another such computa-

tion: the step $\mathbf{w} \leftarrow \mathbf{w} - \eta \nabla L(\mathbf{w})$ is not part of any loss that we want to differentiate. We take one step of gradient descent on the line model with the learning rate $\eta = 0.1$.

Operations inside a `torch.no_grad()` block are not recorded. The update must happen inside such a block: while it records, PyTorch does not allow an in-place change of a *leaf tensor* that requires gradients, that is, of a tensor such as `w` that we marked ourselves and that is not the result of a recorded operation. The gradient in `w.grad` is the $(2, 4)$ computed above.

```
with torch.no_grad():  # nothing in this block is recorded
    w -= 0.1 * w.grad
    L = loss(w)
w, L
```

```
(tensor([ 0.8000, -0.4000], requires_grad=True), tensor(0.4000))
```

The step moves $\mathbf{w}$ to $(0.8, -0.4)$ and lowers the loss from 1 to 0.4, the very numbers that we computed by hand in Section 1.4. Repeating this step is how the models of the next chapter are trained.

The loss computed inside the block has no `grad_fn`: it is not attached to any graph.

1.5.3 Differentiating Programs

The loss of the line model is a fixed formula. Programs often are not: how many times a loop runs, or which branch an `if` statement takes, can depend on the data. Automatic differentiation handles such programs as they are, and, since the backward pass is itself a program, it can differentiate a derivative that it computed.

1.5.3.1 Control Flow

A language model reads sentences of different lengths, so its computation runs for a different number of steps on each input. No single formula describes such a program for all inputs, but none is needed. The framework records the operations that actually run for the input at hand, and these form an ordinary computational graph that the backward pass can walk. Let us try this on a much smaller program with the same two features, a loop whose length and a branch whose choice both depend on the input. The function `f` below doubles its input until the norm of the result reaches 1000, and then returns the result, multiplied by 100 unless its entries have a positive sum.

```
def f(a):
    b = a * 2
    while torch.linalg.vector_norm(b) < 1000:
        b = b * 2
    if b.sum() > 0:
        c = b
    else:
        c = 100 * b
    return c
```

For a scalar input a, the result is $f(a) = ka$ with a factor k that depends on a: a power of two, multiplied by 100 when a is negative. Between the inputs at which k changes, f is

linear, and its derivative is $k = f(a)/a$. We compute both sides of this equation at $a = 0.3$ and at $a = -0.75$.

```
for value in (0.3, -0.75):
    a = torch.tensor(value, requires_grad=True)
    fa = f(a)
    fa.backward()
    print(f'a = {value:5.2f}   f(a) / a = {fa.item() / value:6.0f}   '
          f'gradient = {a.grad.item():6.0f}')
```

```
a =  0.30   f(a) / a =   4096   gradient =   4096
a = -0.75   f(a) / a = 204800   gradient = 204800
```

The two inputs record different graphs (Figure 1.21). For $a = 0.3$, the function doubles 12 times and keeps the result, so $k = 2^{12} = 4096$. For $a = -0.75$, it doubles 11 times and multiplies by 100, so $k = 100 \cdot 2^{11} = 204{,}800$. In both cases the gradient equals $f(a)/a$.

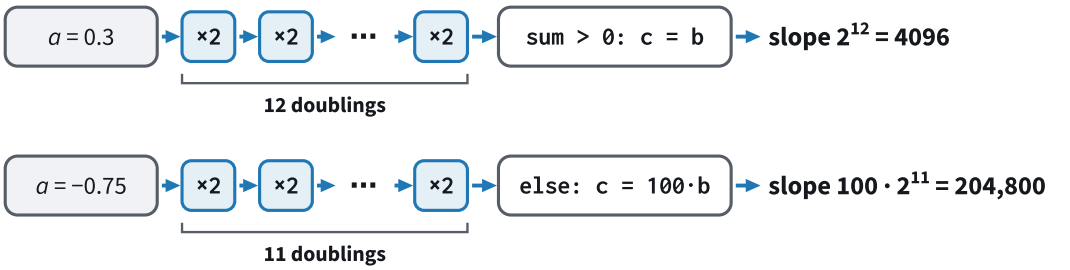


Figure 1.21: The graphs recorded by `f` for two inputs. For $a = 0.3$, `f` doubles a 12 times and the first branch runs; for $a = -0.75$, it doubles a 11 times and the second branch runs. Each graph is linear in a, with its own slope.

Automatic differentiation differentiates the operations that ran, not the function that the program represents. Where the number of doublings changes, at $a = 1000/2^{12} \approx 0.24$ for instance, f jumps and has no derivative. Autograd still returns the slope of the piece that ran. For `f`, the result is the derivative wherever the derivative exists, and a number with no meaning at the jumps.

The backward pass is itself a sequence of tensor operations, so the framework can record it like any other program and differentiate its result once more. This gives second derivatives, such as the curvature of a loss along a direction, and the exercises estimate what they cost.

1.5.4 Forward and Reverse Mode

The rest of this section can be skipped on a first reading. The reader in a hurry needs one result from it: a single backward pass yields the whole gradient of a loss.

Every backward pass in this section started at an output and worked toward the inputs. The chain rule does not prescribe this order. It multiplies the local derivatives along each path through the graph, and the products can be formed from either end. *Reverse mode*, the backward pass that we have used, starts at an output. *Forward mode* starts at an input and carries derivatives forward together with the values. Both compute the same derivatives, at very different costs, and the costs decide which mode trains networks.

1.5.4.1 Rows and Columns of the Jacobian

One traversal of the graph, in either mode, is called a *sweep*; a reverse sweep is a backward pass. Consider a function with n inputs and m outputs and its $m \times n$ Jacobian $\mathbf{J}$. The two kinds of sweep cost about the same, a few evaluations of the function, and each one fills one row or one column of $\mathbf{J}$ (Figure 1.22).

As we saw for vector-valued functions, a reverse sweep starts from a vector $\mathbf{v}$ at the outputs and returns $\mathbf{J}^\top\mathbf{v}$, the transpose of the *vector–Jacobian product* $\mathbf{v}^\top\mathbf{J}$. With $\mathbf{v} = \mathbf{e}_j$, this is row j of $\mathbf{J}$: the gradient of output j with respect to all inputs. Filling the whole Jacobian takes m reverse sweeps, one per row.

A forward sweep starts from a direction $\mathbf{d}$ at the inputs and returns the *Jacobian–vector product* $\mathbf{Jd}$, the rate at which all outputs change as the inputs move along $\mathbf{d}$. With $\mathbf{d} = \mathbf{e}_i$, this is column i: the derivatives of all outputs with respect to input i. Filling the whole Jacobian takes n forward sweeps, one per column.

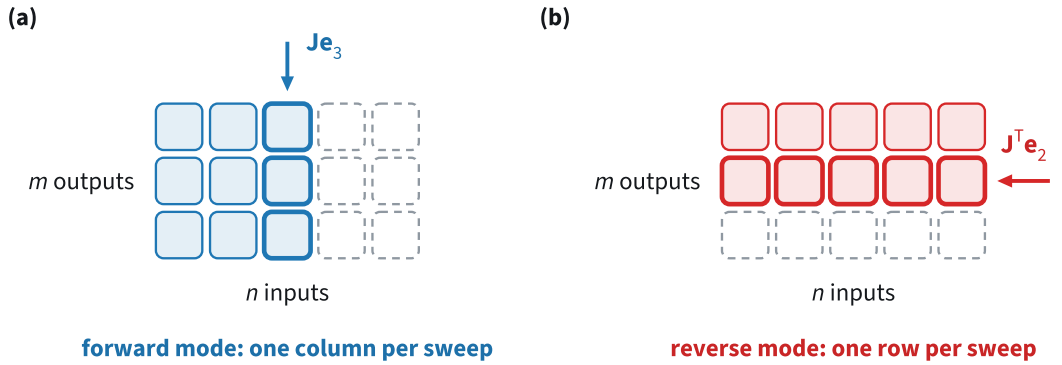


Figure 1.22: Two ways to fill the $m \times n$ Jacobian $\mathbf{J}$ of a function with n inputs and m outputs: (a) forward mode, n sweeps in all; (b) reverse mode, m sweeps in all.

For the errors of the line model, $\mathbf{r}(\mathbf{w}) = \mathbf{X}\mathbf{w} - \mathbf{y}$, the Jacobian is $\mathbf{X}$, with $m = 3$ outputs and $n = 2$ inputs, so reverse mode fills it in three sweeps and forward mode in two.

`torch.func.vjp` and `torch.func.jvp` compute one reverse and one forward sweep, and `torch.func.jacrev` and `torch.func.jacfwd` assemble a whole Jacobian from such sweeps.

1.5.4.2 The Cost of a Gradient

For a loss, the comparison is lopsided. A loss has a single output, $m = 1$, so its Jacobian is one row, the transposed gradient, with an entry for each of the n parameters. Reverse mode computes the whole gradient in one sweep, at a small multiple of the cost of computing the loss, whatever the number of parameters. Forward mode needs one sweep per parameter, n in all, which is no better than finite differences. For a network with $n = 10^8$ parameters, one reverse sweep does the work of 10^8 forward sweeps. This is why deep learning frameworks train networks in reverse mode. Backpropagation is reverse mode applied to a neural network.

Note, however, that reverse mode needs more memory than forward mode, since the forward values that the backward pass uses grow with the number of operations. Forward mode carries one derivative along with each value, so it needs about twice the memory of

evaluating the function, however many operations there are. It is the better choice for functions with few inputs and many outputs.

Both modes are older than deep learning. Forward mode goes back to Wengert (1964), and reverse mode was developed by Linnainmaa (1970) . Section 34.5 derives both modes in detail, including their costs in time and memory.

We have not yet looked inside the backward pass of a network; Section 4.3 does so for a network with one hidden layer. Note also what a gradient cannot tell us. It shows how to lower the loss on the data at hand. Whether a lower loss reflects a pattern that will hold on new data, or only noise in the data we have, is a question of probability and statistics, the subject of the next section.

1.5.5 Summary

Automatic differentiation computes the gradient of a loss from the code that computes the loss, without any derivative written by hand. The framework records the operations of the forward pass as a computational graph and then applies the chain rule backward through the graph, one operation at a time; for the line model this reproduces the gradient $(2, 4)$ that we derived by hand. Detaching a value turns it into a constant for the backward pass, and disabling recording keeps computations such as parameter updates out of the graph. Since the graph records what actually ran, loops and branches need no special treatment, and the backward pass can itself be differentiated. Reverse mode computes a whole gradient in one sweep, which is why it trains the networks of this book.

1.5.6 Exercises

1. code **Detached intermediates.** Let `u = x * x` and `c = u.detach()`, and consider `z = c * c`, `z = u * c`, and `z = u * u`. For each `z`, recompute `u` and `c`, reset `x.grad`, and call `backward` on the sum of the entries of `z`. Before running any code, predict the outcome of each call, including the resulting `x.grad`. Check your predictions, and explain the ratio between the gradients for `z = u * c` and `z = u * u`.

2. code **Manual parameter updates.** Perform the gradient step of this section as `w = w - 0.1 * w.grad`, outside `torch.no_grad()`, and inspect `w.requires_grad`, `w.grad_fn`, `w.is_leaf`, and `w.grad`. Explain what the update has done to the graph, predict what a second forward and backward pass leaves in `w.grad`, and check.

3. code **From scalar to vector input.** Replace the scalar input `a` of the function `f` defined in Section 1.5.3.1 by a random vector of length 5, and differentiate the sum of the entries of `fa = f(a)` with respect to `a`.

    1. Express `f(a)` in terms of `a`, and give the Jacobian of `f` at `a` and the gradient of the sum of the entries of `fa`.
    2. Compute this gradient with automatic differentiation and check whether it still equals `fa / a` elementwise.
    3. Describe the set of inputs `a` at which `f` jumps. Which part of this set has no counterpart for a scalar input?

4. code **Programs versus functions.** Automatic differentiation differentiates the operations that a program executes, not the function that the program computes.

1. The functions $|x|$ and $\sqrt{x^2}$ are equal. Compute the derivative of each at $x = 0$ by automatic differentiation, using the framework's absolute value and square root. Explain both results from the local derivatives of the operations involved. Neither function has a derivative at 0; which of the two results does more harm in training, and why?
2. Write a program that returns x^2 for every input x but whose automatic derivative at $x = 0$ is 1 instead of 0. Explain why automatic differentiation cannot detect the difference.

5. code **Manual tracing.** Let $f(x, y, z) = (x + y) \cdot z$ with $x = -2$, $y = 5$, and $z = -4$.
   1. Draw the computational graph of f with one node per value, compute the forward pass by hand, and trace $\partial f/\partial x$, $\partial f/\partial y$, and $\partial f/\partial z$ backward from f through the multiplication and addition nodes (reverse mode).
   2. Compute the same three derivatives in forward mode, propagating the derivative with respect to one input at a time from the inputs to f. Compare the number of sweeps each mode needs and the intermediate values each must keep in memory.
   3. Verify every value using automatic differentiation.

   Adapted from Stanford CS231n, "An Exercise in Backpropagation".

6. **Cost of second derivatives.** ⊠ The counting argument in Section 1.5.4 puts the cost of the gradient of a scalar loss with n parameters at one reverse sweep. Use the same argument to estimate, in units of function evaluations, the cost of the full Hessian (the $n \times n$ matrix of second derivatives) and the cost of a single Hessian–vector product. Explain the difference between the two costs.

1.6 Probability and Statistics

The previous sections gave us the tools to compute a prediction, a loss, and its gradient. Once a model is trained, we measure how well it does, and the measurement depends on which examples we happened to use. A classifier that labels 230 of a minibatch of 256 images correctly has an accuracy of 0.90 on that minibatch, and on the next minibatch the same model may score 0.87. Before we conclude that one model is better than another, or that a change to the training code helped, we need to know how much such numbers fluctuate when nothing has changed.

Such questions are much older than machine learning. Probability theory began in the seventeenth century as a study of games of chance, and its simplest example is still a coin. Suppose that a coin is tossed 100 times and produces 56 heads. Is the coin biased? The count alone does not tell us. Even a fair coin rarely produces exactly 50 heads in 100 tosses, so some deviation from 50 is expected, and the question is whether a deviation of 6 is larger than chance alone would produce.

Probability answers the forward question: given a model of a random process, such as a fair coin, how likely is each outcome? *Statistics* answers the inverse question: given observations, what can we infer about the process that produced them? Together they let us judge whether a number computed from data reflects a stable pattern or sampling noise.

The coin question is easy to ask, and it takes us most of the section to answer. We first simulate coin tosses to see the noise. We then state the rules for computing with random variables, among them the independence of the tosses, and measure how much an average

of random outcomes fluctuates, which settles the question. The last part of the section turns to what one variable tells us about another, and uses it to interpret a medical test.

```
import random
from torch.distributions.multinomial import Multinomial
```

1.6.1 Sampling

1.6.1.1 Simulated Coin Tosses

If a coin is *fair*, heads and tails are equally likely. We express this by saying that the probability of heads is one half, written $P(\text{heads}) = 0.5$. A probability is a number between 0 and 1 attached to an outcome, or more generally to a set of outcomes. A probability of 1 means certainty, as for heads on a two-headed coin, and a probability of 0 means impossibility. The probability 0.5 is a property of the coin and of how it is tossed, not of any particular sequence of tosses.

To see what a fair coin produces, we simulate it. Python's `random.random` returns a number drawn uniformly from the interval $[0, 1)$: the probability that it falls into a subinterval $[a, b]$ is $b - a$. A draw therefore exceeds 0.5 with probability 0.5, and comparing it with 0.5 simulates one toss.

```
num_tosses = 100
heads = sum([random.random() > 0.5 for _ in range(num_tosses)])
tails = num_tosses - heads
print("heads, tails: ", [heads, tails])
```

```
heads, tails:  [45, 55]
```

Running the cell again gives different counts, and they are usually not 50 and 50. Deep learning frameworks provide samplers for any random outcome with finitely many values, such as the toss of a coin or the roll of a die. The *multinomial* distribution describes how often each outcome occurs in n draws that do not influence one another, such as n tosses of a coin. We pass n and the vector of probabilities, here $(0.5, 0.5)$ for heads and tails, and receive the count of each outcome.

```
fair_probs = torch.tensor([0.5, 0.5])
Multinomial(100, fair_probs).sample()
```

```
tensor([48., 52.])
```

1.6.1.2 Frequencies and Probabilities

Dividing the counts by the number of tosses gives the *frequency* of each outcome. We write $p = P(\text{heads})$ for the probability of heads and $\hat{p}$ for its frequency in n tosses; the hat marks a quantity computed from data. To see how much the frequencies vary, we repeat the 100 tosses five times.

```
Multinomial(100, fair_probs).sample((5,)) / 100
```

```
tensor([[0.5000, 0.5000],
        [0.4500, 0.5500],
        [0.5000, 0.5000],
        [0.4900, 0.5100],
        [0.5800, 0.4200]])
```

The frequencies vary from one repetition to the next, while the probability stays at 0.5. A quantity computed from data to estimate an unknown quantity is called an *estimator*, and $\hat{p}$ is the natural estimator of p.

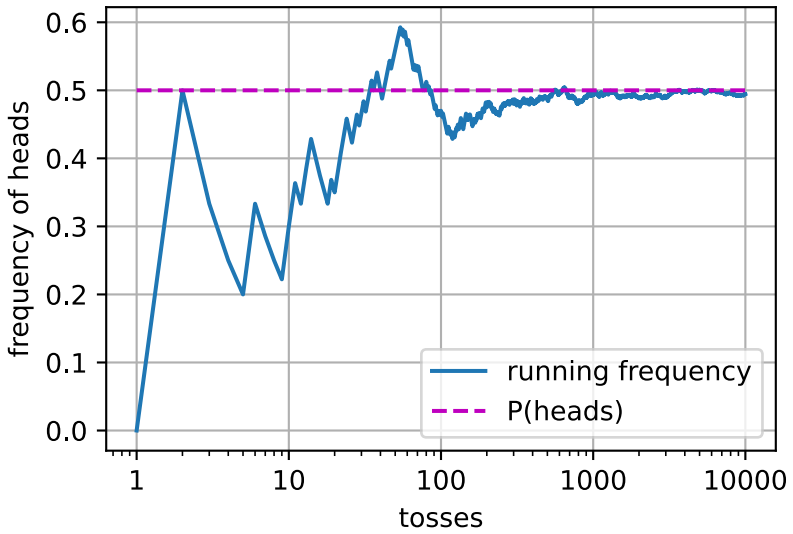
Had we not written the simulation ourselves, a single run could not tell us whether the coin is slightly unfair or whether the deviation from 0.5 is due to chance. More tosses help: with $10{,}000$ tosses, the frequencies typically lie much closer to 0.5.

```
Multinomial(10000, fair_probs).sample() / 10000
```

```
tensor([0.4959, 0.5041])
```

How does a single estimate behave as the tosses accumulate? After each toss, we divide the number of heads so far, which the cumulative sum `cumsum` provides, by the number of tosses so far. The horizontal line marks $P(\text{heads}) = 0.5$.

```
n = 10000
tosses = (torch.rand(n) < 0.5).to(torch.float32)  # 1 is heads, 0 is tails
estimates = tosses.cumsum(dim=0) / torch.arange(1, n + 1)
d2l.plot(torch.arange(1, n + 1), [estimates, torch.full((n,), 0.5)],
         'tosses', 'frequency of heads',
         legend=['running frequency', 'P(heads)'], xscale='log',
         figsize=(4.5, 3))
```



The running frequency swings widely at first. The swings shrink as tosses accumulate, and after a few thousand tosses the frequency stays close to 0.5. That the frequency approaches the probability as the number of tosses grows is the *law of large numbers*, which Section 36.5

states precisely and proves. Unfortunately, the law does not say how fast the frequency settles, and that is what the coin question needs: how far from 0.5 can the frequency of 100 tosses lie before we should suspect the coin? We cannot answer this yet. We first need a precise language for random outcomes, and then a way to measure how much their averages fluctuate.

1.6.2 Events and Random Variables

1.6.2.1 Sample Spaces and Events

The set of possible outcomes of a random experiment is its *sample space* $\mathcal{S}$. For one toss of a coin, $\mathcal{S} = \{\text{heads}, \text{tails}\}$. For one roll of a die, $\mathcal{S} = \{1, 2, 3, 4, 5, 6\}$. For two tosses of a coin, $\mathcal{S}$ holds the four pairs (heads, heads), (heads, tails), (tails, heads), and (tails, tails). An *event* is a subset of the sample space, and it occurs if the outcome lies in it. For a die, "an even number" is the event $\mathcal{A} = \{2, 4, 6\}$ and "at least 5" is the event $\mathcal{B} = \{5, 6\}$. A roll of 6 makes both events occur; a roll of 5 makes only $\mathcal{B}$ occur.

A probability P assigns a number to every event. It must obey three rules, the axioms of Kolmogorov (1933):

- $P(\mathcal{A}) \geq 0$ for every event $\mathcal{A}$;
- $P(\mathcal{S}) = 1$;
- if the events $\mathcal{A}_1, \mathcal{A}_2, \ldots$ are *mutually exclusive*, that is, $\mathcal{A}_i \cap \mathcal{A}_j = \emptyset$ for $i \neq j$, then the probability that one of them occurs is the sum of their probabilities, $P(\bigcup_i \mathcal{A}_i) = \sum_i P(\mathcal{A}_i)$.

The list is short, and it suffices: every other rule of probability follows from these three. An event $\mathcal{A}$ and its *complement* $\mathcal{A}'$, the outcomes not in $\mathcal{A}$, are mutually exclusive and together make up $\mathcal{S}$, so $P(\mathcal{A}') = 1 - P(\mathcal{A})$. For $\mathcal{A} = \mathcal{S}$ this gives $P(\emptyset) = 0$. When two events overlap, adding their probabilities counts the outcomes that lie in both of them twice (Figure 1.23). The union $\mathcal{A} \cup \mathcal{B}$ splits into the mutually exclusive pieces $\mathcal{A}$ and $\mathcal{B} \setminus \mathcal{A}$, and $P(\mathcal{B} \setminus \mathcal{A}) = P(\mathcal{B}) - P(\mathcal{A} \cap \mathcal{B})$ for the same reason. Together these give the *inclusion–exclusion* rule

$$P(\mathcal{A} \cup \mathcal{B}) = P(\mathcal{A}) + P(\mathcal{B}) - P(\mathcal{A} \cap \mathcal{B}). \tag{1.37}$$

For a fair die, the probability of an even number or a number of at least 5 is $3/6 + 2/6 - 1/6 = 2/3$, and indeed the union $\{2, 4, 5, 6\}$ holds four of the six outcomes.

1.6.2.2 Random Variables

Often we care about a number computed from the outcome rather than about the outcome itself: the number of heads in 100 tosses, the sum of two dice, or whether a classifier labels an image correctly. A *random variable* is such a function from the sample space to a set of values. It can be much coarser than the outcome. The number of heads in 100 tosses takes 101 values, while the sample space holds 2^{100} sequences of tosses, a number with 31 digits. Each value v of a random variable X defines an event, the set of outcomes for which $X = v$, so $P(X = v)$ is defined. A range of values is an event too: $P(1 \leq X \leq 3)$ is the probability that X takes a value between 1 and 3. The table of the probabilities $P(X = v)$ of all values v is the *distribution* of X, also called its *probability mass function*, and we write it as $P(X)$.

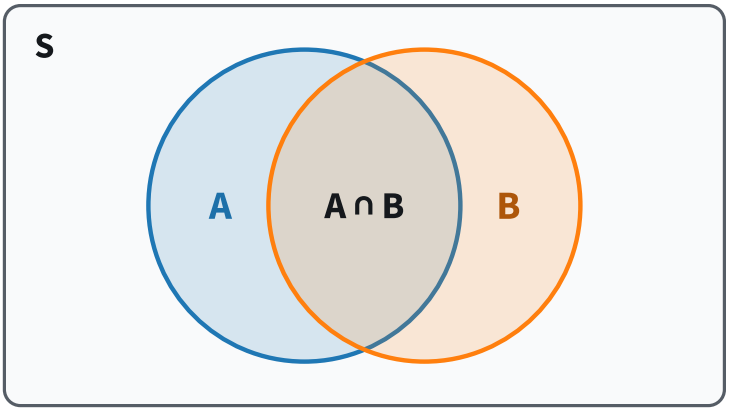


Figure 1.23: Two events $\mathcal{A}$ and $\mathcal{B}$ in a sample space $\mathcal{S}$. The sum $P(\mathcal{A}) + P(\mathcal{B})$ counts the overlap $\mathcal{A} \cap \mathcal{B}$ twice.

1.6.2.3 Discrete and Continuous Variables

The random variables so far are *discrete*: they take finitely or countably many values, each with its own probability. Many quantities are *continuous* instead, such as the height of a person drawn at random from a population. With fine enough measurements no two people have the same height, so the probability that someone is exactly 1.80139278 meters tall is zero, even though such a height is possible. The useful question is whether a height falls into an interval, say between 1.79 and 1.81 meters. A continuous variable has a *probability density* $p(x)$ (unrelated to the probability p of heads), and the probability of an interval is the area under the density over it (Figure 1.24),

$$P(a \le X \le b) = \int_a^b p(x)\,dx. \tag{1.38}$$

Densities, and the sense in which a possible event can have probability zero, are developed in Section 36.1. In this section we work with discrete variables. Every sum over the values of a variable becomes an integral when the variable has a density.

1.6.2.4 Joint and Marginal Distributions

Each toss is one random variable, and to say how a hundred tosses combine we must be able to speak of several variables at once. So must most of machine learning, whose questions involve the pixels of an image and its label, or the symptoms of a patient and the disease. When we draw an individual from a population, each variable takes a value.

The *joint distribution* of two random variables X and Y assigns a probability to every combination of values, $P(X = x, Y = y)$, where the comma means "and". Take a day drawn at random, and let X record whether it rains and Y whether we carry an umbrella. The joint distribution is a table with one entry per combination:

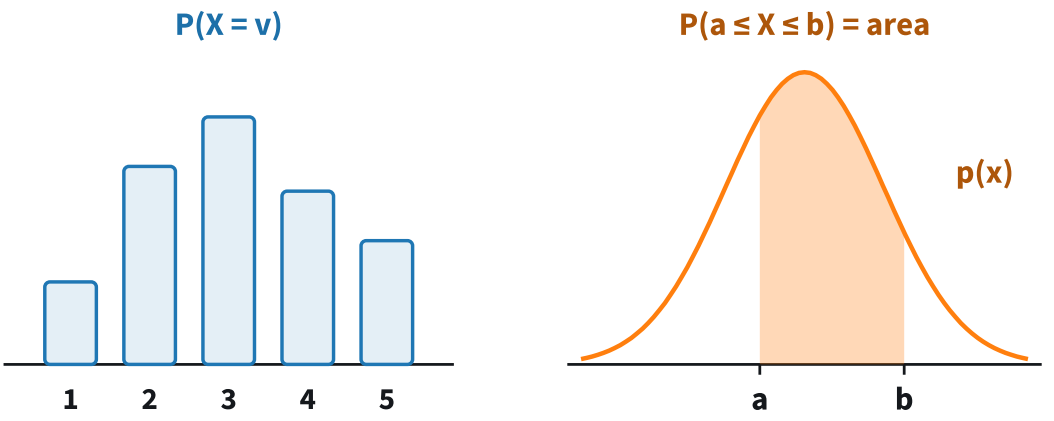


Figure 1.24: A discrete random variable puts probability on individual values (left). A continuous random variable has a density, and the probability of an interval is the area under the density (right).

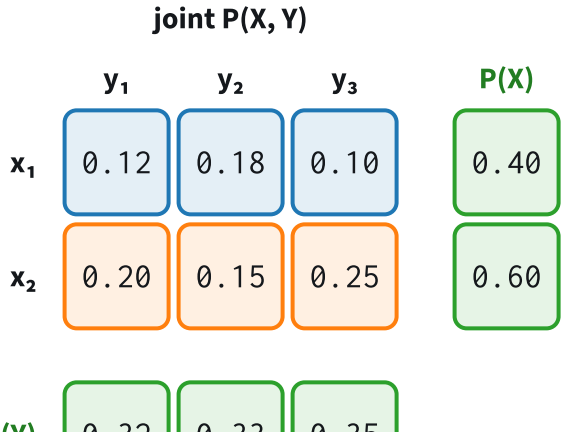
$P(X, Y)$	umbrella	no umbrella
rain	0.25	0.05
no rain	0.10	0.60

The four entries sum to 1, since every day falls into exactly one cell. Figure 1.25 shows a second such table, for a variable X with two values and a variable Y with three, with no story attached.

The joint distribution contains everything there is to know about the two variables. In particular, it determines the distribution of each variable on its own, its *marginal distribution*. The event $X = x$ occurs together with exactly one value of Y, so

$$P(X = x) = \sum_y P(X = x, Y = y). \tag{1.39}$$

Summing each row of the table gives the marginal distribution of X, and summing each column gives that of Y. It rains on $P(\text{rain}) = 0.25 + 0.05 = 0.30$ of the days, and we carry the umbrella on $0.25 + 0.10 = 0.35$ of them; in the figure, $P(X = x_2) = 0.20 + 0.15 + 0.25 = 0.60$. Removing a variable by summing over its values is called *marginalization*. As for a single variable, $P(X, Y)$ denotes the whole table, and an equation between distributions, such as $P(X, Y) = P(X)P(Y)$, means that it holds for all values of the variables.



1.6.2.5 Independence

Two random variables are *independent*, written $X \perp Y$, if their joint distribution factorizes,

$$P(X, Y) = P(X)P(Y). \tag{1.40}$$

Two separate tosses of a coin are independent, since the first toss says nothing about the second. Rain and umbrellas are not: $P(\text{rain}, \text{umbrella}) = 0.25$, while the product of the marginals is $0.30 \cdot 0.35 \approx 0.11$. Independence is the notion that the coin question needs, since independence between successive draws of data is what makes averages reliable, as we will see when we measure how much they fluctuate. Dependence, on the other hand, is often what we want to learn: we predict a disease from symptoms because the two are *not* independent.

1.6.3 Expectation and Variance

1.6.3.1 Expectation

A distribution lists the probability of every value. Often we need a single number that summarizes it, such as the value to expect on average. Consider an investment that is lost entirely with probability 0.5, doubles with probability 0.4, and grows tenfold with probability 0.1. Weighting each return by its probability gives the expected return $0.5 \cdot 0 + 0.4 \cdot 2 + 0.1 \cdot 10 = 1.8$. In general, the *expectation* or *mean* of a random variable X is the average of its values weighted by their probabilities,

$$\mathbb{E}[X] = \sum_{x} x\, P(X = x). \tag{1.41}$$

If we place a weight $P(X = x)$ at each position x on a line, the expectation is the point at which the line balances. The expectation of a function of X is computed in the same way, $\mathbb{E}[f(X)] = \sum_x f(x) P(X = x)$.

For a coin toss, let $X = \mathbf{1}(\text{heads})$ be the *indicator* of heads: $X = 1$ if the toss comes up heads and $X = 0$ otherwise. Then $\mathbb{E}[X] = 1 \cdot p + 0 \cdot (1 - p) = p$, so the expectation of an indicator is the probability of its event.

The property of expectations that we will use most is that they are *linear*: for any random variables X and Y and numbers a and b, $\mathbb{E}[aX + bY] = a\mathbb{E}[X] + b\mathbb{E}[Y]$. This follows by writing $\mathbb{E}[aX + bY]$ as a sum over the joint distribution of X and Y and splitting the sum (Section 36.1 gives the details). Note that it holds whether or not X and Y are independent. Let X_i be the indicator of heads on toss i, so that the frequency of heads is their average. Linearity gives

$$\mathbb{E}[\hat{p}] = \mathbb{E}\left[\frac{1}{n}\sum_{i=1}^{n} X_i\right] = \frac{1}{n}\sum_{i=1}^{n} \mathbb{E}[X_i] = p. \tag{1.42}$$

On average, the frequency equals the probability that it estimates; an estimator with this property is called *unbiased*. This is reassuring, but it does not answer the coin question. Unbiasedness says nothing about how far a single estimate strays from p. For that we need a second summary.

1.6.3.2 Variance

The investment above has an expected return of 1.8, yet half of the time it loses everything. To measure how far values tend to fall from the mean $\mu = \mathbb{E}[X]$, we cannot average the deviation $X - \mu$ itself, since by linearity $\mathbb{E}[X - \mu] = \mathbb{E}[X] - \mu = 0$: positive and negative deviations cancel. Averaging the squared deviation instead gives the *variance*,

$$\operatorname{Var}(X) = \mathbb{E}\left[(X - \mu)^2\right] = \mathbb{E}[X^2] - \mu^2. \tag{1.43}$$

The second form follows by expanding $(X - \mu)^2 = X^2 - 2\mu X + \mu^2$ and taking the expectation of each term.

The variance is measured in squared units. Its square root, the *standard deviation* $\sigma = \sqrt{\operatorname{Var}(X)}$, has the units of X and measures the typical distance between X and its mean. For the investment, $\mathbb{E}[X^2] = 0.4 \cdot 2^2 + 0.1 \cdot 10^2 = 11.6$, so $\operatorname{Var}(X) = 11.6 - 1.8^2 = 8.36$ and $\sigma \approx 2.9$, larger than the mean itself.

When we have only n numbers and not the distribution, as for the columns in Section 1.2, we estimate the variance by averaging their squared deviations from their own average. The numbers lie closer to their own average than to the true mean μ, so dividing the sum of squares by n underestimates the variance on average. Dividing by $n - 1$, as pandas does, makes the estimate unbiased.

For the indicator of heads, $X^2 = X$, so

$$\operatorname{Var}(X) = p - p^2 = p(1 - p). \tag{1.44}$$

A toss is most uncertain for a fair coin, where the variance reaches its largest value, $1/4$. The variance is zero for a coin that always lands the same way. Shifting a random variable by a constant b does not change its deviations from the mean, and scaling it by a scales the deviations by a and their squares by a^2, so $\operatorname{Var}(aX + b) = a^2 \operatorname{Var}(X)$.

1.6.3.3 Sums of Independent Variables

The frequency of heads is an average of many tosses, so we need the variance of a sum. Write $\mu_X = \mathbb{E}[X]$ and $\mu_Y = \mathbb{E}[Y]$. Expanding the square in the definition of variance gives

$$\operatorname{Var}(X + Y) = \mathbb{E}\left[(X - \mu_X + Y - \mu_Y)^2\right] = \operatorname{Var}(X) + \operatorname{Var}(Y) + 2\operatorname{Cov}(X, Y), \tag{1.45}$$

where the cross term

$$\operatorname{Cov}(X, Y) = \mathbb{E}\left[(X - \mu_X)(Y - \mu_Y)\right] \tag{1.46}$$

is the *covariance* of X and Y. The covariance is positive when X and Y tend to lie on the same side of their means, and negative when they tend to lie on opposite sides (Figure 1.26).

The size of the covariance depends on the units of X and Y. Dividing by both standard deviations gives the *correlation* $\rho = \operatorname{Cov}(X, Y)/(\sigma_X \sigma_Y)$, which has no units and always lies between -1 and 1.

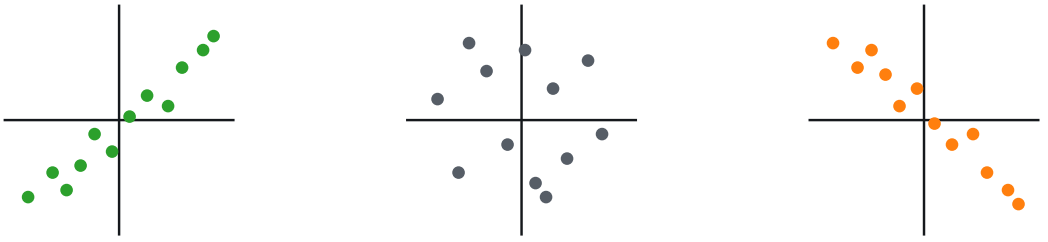


Figure 1.26: Samples of three pairs of random variables, with positive, near-zero, and negative covariance.

For independent variables the covariance vanishes. Since the joint distribution factorizes, the expectation of a product of a function of X and a function of Y factorizes too, $\mathbb{E}[f(X)g(Y)] = \mathbb{E}[f(X)]\,\mathbb{E}[g(Y)]$, so $\mathrm{Cov}(X,Y) = \mathbb{E}[X-\mu_X]\,\mathbb{E}[Y-\mu_Y] = 0$. Hence, for independent X and Y,

$$\mathrm{Var}(X+Y) = \mathrm{Var}(X) + \mathrm{Var}(Y). \tag{1.47}$$

This is the key step toward the coin. Variances of independent variables add, while standard deviations do not. The sum of n independent variables that each have variance σ^2 has variance $n\sigma^2$ and therefore standard deviation $\sqrt{n}\,\sigma$, not $n\sigma$, because deviations of opposite signs partly cancel.

The converse does not hold: zero covariance does not imply independence. Let X be uniform on $\{-1,0,1\}$ and $Y = X^2$. Then $\mathbb{E}[X] = 0$ and $\mathbb{E}[XY] = \mathbb{E}[X^3] = 0$, so $\mathrm{Cov}(X,Y) = 0$, yet Y is determined by X. Covariance detects only a tendency to move together along a line.

One last definition, which the coin does not need and later chapters do. For a random vector $\mathbf{x}$ with mean μ, the variances and covariances of all coordinates form the *covariance matrix*

$$\Sigma = \mathbb{E}\left[(\mathbf{x}-\mu)(\mathbf{x}-\mu)^{\top}\right], \tag{1.48}$$

whose diagonal holds the variances of the coordinates and whose entry Σ_{ij} is the covariance of x_i and x_j.

1.6.4 Estimation Error

1.6.4.1 The Standard Error of a Frequency

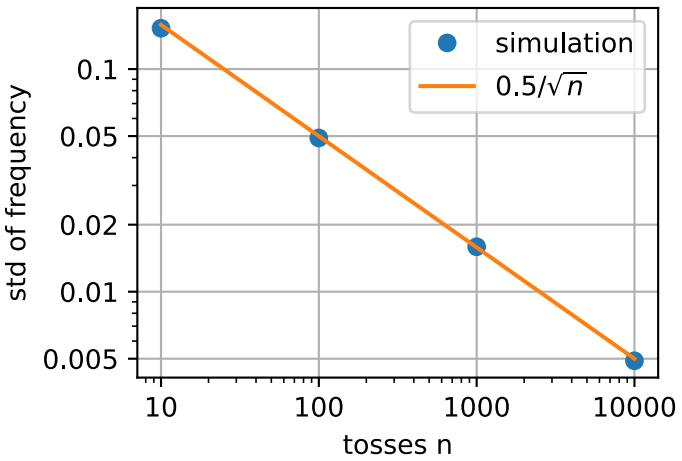
Everything is now in place to compute how much the frequency of heads fluctuates. The frequency $\hat{p} = \frac{1}{n}\sum_{i=1}^{n} X_i$ averages n independent indicators, each with variance $p(1-p)$. Their sum therefore has variance $np(1-p)$, and dividing the sum by n multiplies its variance by $1/n^2$:

$$\mathrm{Var}(\hat{p}) = \frac{p(1-p)}{n}. \tag{1.49}$$

The standard deviation of an estimate is called its *standard error*. For the frequency of heads it is $\sqrt{p(1-p)/n}$, which for a fair coin equals $0.5/\sqrt{n}$: 0.05 after 100 tosses and 0.005 after $10{,}000$. The error falls only with the square root of the number of tosses, which is slow. Halving it takes four times as many tosses, and reducing it tenfold takes a hundred times as many. When p is unknown, we substitute its estimate and use $\sqrt{\hat{p}(1-\hat{p})/n}$, which the data alone determine.

To check the formula, we repeat the experiment many times. For each n, we compute 1000 independent frequencies, each from n tosses of a fair coin, and measure their standard deviation.

```
ns = [10, 100, 1000, 10000]
stds = [float((torch.rand(1000, n) < 0.5).to(torch.float32)
               .mean(dim=1).std()) for n in ns]
d2l.plot(ns, [stds, [0.5 / n ** 0.5 for n in ns]], 'tosses n',
         'std of frequency', legend=['simulation', r'$0.5/\sqrt{n}$'],
         xscale='log', yscale='log', fmts=('o', '-'))
```



On log–log axes, the measured standard deviations lie on the predicted line $0.5/\sqrt{n}$, whose slope is $-1/2$. Note that the derivation used nothing about the coin except the variance of a single toss and the independence of the tosses. The average of n independent draws of any random variable with variance σ^2 therefore has variance σ^2/n and standard error $\sigma/\sqrt{n}$. This rate governs every average in this book, from the accuracy of a model on a test set to the gradient computed on a minibatch of examples.

1.6.4.2 Testing for Bias

At last we can return to the 56 heads in 100 tosses. The frequency 0.56 lies 0.06 from 0.5, which is 1.2 standard errors. This does not sound like much: a fair coin produces deviations of about one standard error all the time. To see how often it produces one at least this large,

we simulate 1000 experiments with a fair coin and count the fraction of them whose number of heads lies at least as far from $n/2$ as the observed count. We do this for 56 heads in 100 tosses and for the same frequency in $10{,}000$ tosses.

```
def fraction_as_far(heads, n, num_experiments=1000):
    """Fraction of fair-coin experiments at least as far from n/2 as heads."""
    counts = (torch.rand(num_experiments, n) < 0.5).sum(dim=1)
    far = (counts - n / 2).abs() >= abs(heads - n / 2)
    return far.to(torch.float32).mean()

fraction_as_far(56, 100), fraction_as_far(5600, 10000)
```

```
(tensor(0.2820), tensor(0.))
```

Roughly one in four fair-coin experiments with 100 tosses ends at least as far from 50 heads as 56 does. This fraction estimates what statisticians call a *p-value*: the probability, if the coin is fair, of a result at least as extreme as the one observed. An outcome that a fair coin produces one time in four gives no evidence that the coin is biased. Nor is it evidence that the coin is fair: 56 heads are just as consistent with $p = 0.55$. They do rule out $p = 0.3$, which lies more than five standard errors away.

The same frequency after $10{,}000$ tosses, 5,600 heads, lies $0.06/0.005 = 12$ standard errors from 0.5, and none of the 1000 simulated experiments lands that far out. A frequency of 0.56 is therefore consistent with a fair coin after 100 tosses and strong evidence of bias after $10{,}000$. What decides is the size of the deviation measured in standard errors.

How many standard errors are too many? For every distribution, a random variable lies k or more standard deviations from its mean with probability at most $1/k^2$, a result known as *Chebyshev's inequality* and proved in Section 36.1. For $k = 12$ the bound is below 1%, but for $k = 2$ it is a rather crude 25%. Averages are much better behaved. The average of many independent draws is distributed approximately like the normal distribution of Section 2.1.3 (the central limit theorem, Section 36.3), which lies within two standard deviations of its mean about 95% of the time. Hence the rule of thumb, which we state without proof: a deviation of less than two standard errors is unremarkable, and one of three or more is hard to blame on chance.

The same reasoning applies to measuring a model. Suppose that a classifier labels each test image correctly with probability 0.9, independently of the other images. Its accuracy on a minibatch of 256 images is then a frequency with standard error $\sqrt{0.9 \cdot 0.1/256} \approx 0.019$, so two minibatches routinely differ by a few percentage points even when nothing has changed. On a test set of $10{,}000$ images, the standard error drops to 0.003, and that of the difference between two independently measured accuracies is about 0.004, since the variances add. A difference of one percentage point between two models is then more than two standard errors and unlikely to be noise. The exercises return to this comparison.

1.6.4.3 Aleatoric and Epistemic Uncertainty

The standard error measures our uncertainty about p, and it shrinks as tosses accumulate. Our uncertainty about the next toss does not shrink: even if we knew that $p = 0.5$ exactly, the next toss would come up heads only half of the time. The two kinds of uncertainty have names (Der Kiureghian and Ditlevsen 2009). *Epistemic uncertainty* is uncertainty about the model, such as the value of p, and more data reduces it, at the slow rate of $1/\sqrt{n}$. *Aleatoric*

uncertainty is randomness in the outcome itself, which remains even when the model is known exactly. A classifier trained on ever more images becomes more certain of the probabilities it assigns, but for an image that is ambiguous, such as a blurry photograph that could show a shirt or a coat, those probabilities stay spread out. The aleatoric part sets a floor that no amount of data can remove.

1.6.5 Conditional Probability

Knowing the value of one variable can change what we expect of another. A patient who has trouble breathing and has lost the sense of smell is more likely to have COVID-19 than one without these symptoms. This part makes such statements precise and applies them to a medical test.

1.6.5.1 Conditional Distributions

Suppose that we learn that $X = x_2$. Only the outcomes in the second row of the table remain possible, and their probabilities must be rescaled so that they sum to 1. This gives the *conditional probability* of $Y = y$ given $X = x$,

$$P(Y = y \mid X = x) = \frac{P(X = x, Y = y)}{P(X = x)},$$

which is defined whenever $P(X = x) > 0$. In the umbrella table, the rain row $(0.25, 0.05)$ becomes $(0.83, 0.17)$: on a rainy day we carry the umbrella with probability 0.83, against 0.35 on a day about which we know nothing. In the figure, the second row $(0.20, 0.15, 0.25)$ becomes $(0.33, 0.25, 0.42)$. For each fixed x, the conditional probabilities form a distribution over the values of Y and obey all three axioms. Independence, in these terms, means that learning the value of Y does not change the distribution of X: $P(X \mid Y = y) = P(X)$ whenever $P(Y = y) > 0$.

Multiplying both sides of the definition by $P(X = x)$ gives the *multiplication rule*

$$P(X, Y) = P(Y \mid X)P(X). \tag{1.50}$$

The probability that both events occur is the probability of the first times the probability of the second once the first is known.

1.6.5.2 Bayes' Theorem

Often one conditional probability is easy to obtain while we need the other. A medical study measures how often a test is positive among patients who have a disease, $P(\text{positive} \mid \text{disease})$. A doctor facing a positive result needs $P(\text{disease} \mid \text{positive})$. The multiplication rule connects the two. Writing the joint distribution in both orders, $P(X, Y) = P(Y \mid X)P(X) = P(X \mid Y)P(Y)$, and dividing by $P(Y)$ gives *Bayes' theorem* (Bayes 1763),

$$P(X \mid Y) = \frac{P(Y \mid X)P(X)}{P(Y)}. \tag{1.51}$$

For a result that takes one line of algebra, Bayes' theorem is remarkably useful: it reverses the direction of conditioning. The denominator needs no separate measurement: marginalizing the multiplication rule over X gives $P(Y) = \sum_x P(Y \mid X = x)P(X = x)$, so $P(Y \mid X)$ and $P(X)$ determine everything.

In inference, X is a hypothesis H about the world and Y is the data D that we observe. The *prior* $P(H)$ says how plausible the hypothesis is before we see the data, the *likelihood* $P(D \mid H)$ says how well the hypothesis explains the data, and the *posterior* $P(H \mid D)$ combines the two. Since the denominator $P(D)$ is the same for every hypothesis, Bayes' theorem reads

$$P(H \mid D) \propto P(D \mid H)P(H). \tag{1.52}$$

The posterior is proportional to the likelihood times the prior, and dividing by the sum over all hypotheses makes it sum to 1. Probabilities used in this way express degrees of belief rather than frequencies of repeatable events. The rules for calculating with them are the same. We put the theorem to work on a medical test below, after one more notion that the example needs.

1.6.5.3 Conditional Independence

Independence can also hold within a conditional distribution. The variables X and Y are *conditionally independent* given Z, written $X \perp Y \mid Z$, if $P(X, Y \mid Z) = P(X \mid Z)P(Y \mid Z)$. Conditioning can remove dependence, and it can create it (Figure 1.27). Among elementary school students, shoe size and reading level are strongly related, because both grow with age, their common cause. Among students of the same age the relation disappears. A chain behaves similarly. If X affects Z only through Y, as yesterday's weather might affect tomorrow's only through today's, then knowing Y makes X irrelevant for Z.

In the opposite direction, broken bones and lung cancer may be independent in the general population, yet among hospital patients they are related. A patient with a broken bone already has a reason to be in the hospital, and is therefore less likely to be there because of lung cancer. Conditioning on a common effect, here being in the hospital, makes its independent causes dependent. This is called *explaining away*.

1.6.5.4 Example: Diagnostic Testing

A doctor administers an HIV test to a patient. The test never misses an infection, but it reports a *false positive* for 1% of healthy patients. We write $H \in \{0, 1\}$ for the patient's HIV status (1 for infected) and $D_1 \in \{0, 1\}$ for the test result (1 for positive). The test is described by conditional probabilities:

$P(D_1 \mid H)$	$H = 1$ (HIV)	$H = 0$ (healthy)
test positive, $D_1 = 1$	1.00	0.01
test negative, $D_1 = 0$	0.00	0.99

Each column is the conditional distribution of D_1 given one value of H, so the columns sum to 1, while the rows need not. Suppose that the test comes back positive. What is the probability $P(H = 1 \mid D_1 = 1)$ that the patient has HIV? The answer depends on how

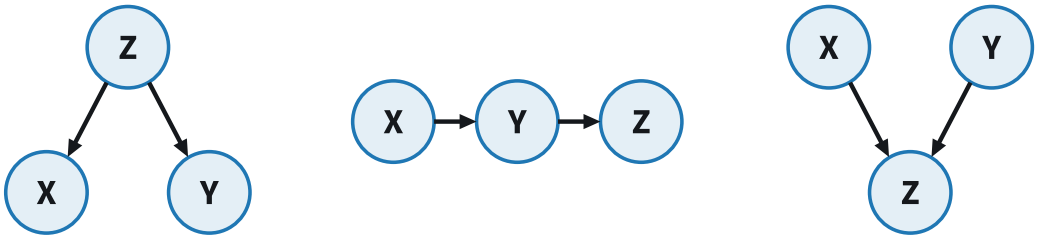


Figure 1.27: Conditioning changes dependence. A common cause Z makes X and Y dependent, and observing Z removes the dependence. In a chain, X and Z are independent given Y. A common effect Z makes its independent causes dependent once Z is observed.

common the disease is, since that determines how many false alarms compete with the true detections. Assume that 0.15% of the population is infected, $P(H = 1) = 0.0015$. The denominator of Bayes' theorem follows by marginalizing over H:

$$\begin{aligned} P(D_1 = 1) &= P(D_1 = 1 \mid H = 0)P(H = 0) + P(D_1 = 1 \mid H = 1)P(H = 1) \\ &= 0.01 \cdot 0.9985 + 1 \cdot 0.0015 = 0.011485. \end{aligned}$$

Bayes' theorem then gives

$$P(H = 1 \mid D_1 = 1) = \frac{P(D_1 = 1 \mid H = 1)P(H = 1)}{P(D_1 = 1)} = \frac{0.0015}{0.011485} = 0.1306. \quad (1.53)$$

A patient who tests positive has HIV with a probability of only 13%, even though the test never misses an infection and errs on just 1% of healthy patients. Figure 1.28 shows the reason in counts. Among 10,000 people, about 15 are infected and all of them test positive, but so do about 100 of the 9,985 healthy people. Because the disease is rare, the false positives outnumber the true ones.

A doctor would therefore order a second test. It is less accurate than the first:

$P(D_2 \mid H)$	$H = 1$ (HIV)	$H = 0$ (healthy)
test positive, $D_2 = 1$	0.98	0.03
test negative, $D_2 = 0$	0.02	0.97

The second test comes back positive too. To combine the two results, we assume that the tests are conditionally independent given H, their common cause in the sense of Figure 1.27: once the patient's status is known, the result of one test says nothing about the result of the other. Then the likelihoods of the two results multiply,

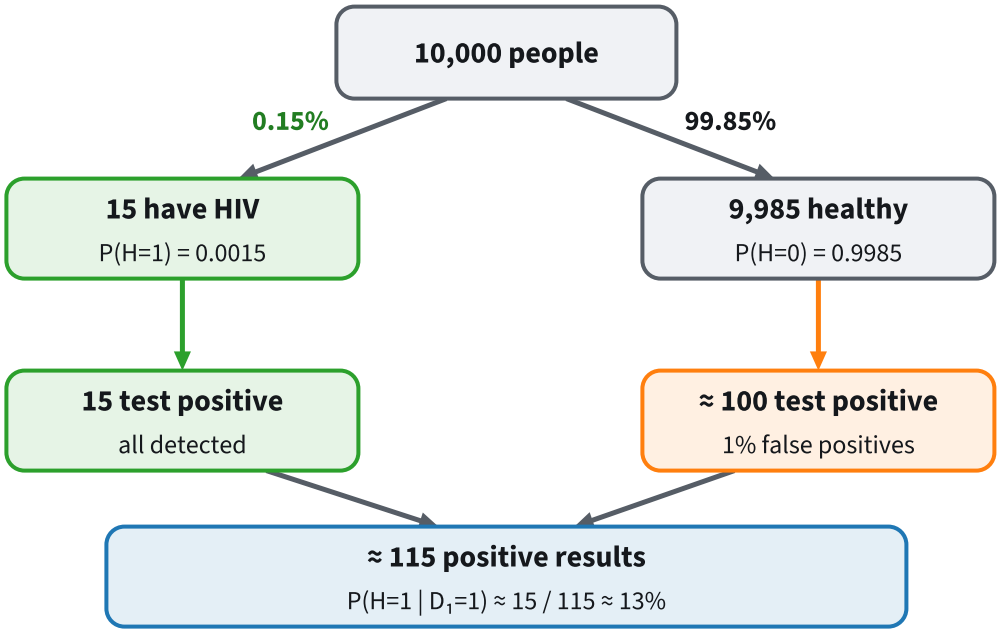


Figure 1.28: The HIV test applied to 10,000 people. The 15 infected people test positive, and so do about 100 of the 9,985 healthy people, so only 15 of about 115 positive results are correct.

$$P(D_1 = 1, D_2 = 1 \mid H = 0) = P(D_1 = 1 \mid H = 0)P(D_2 = 1 \mid H = 0) = 0.01 \cdot 0.03 = 0.0003,$$
$$P(D_1 = 1, D_2 = 1 \mid H = 1) = P(D_1 = 1 \mid H = 1)P(D_2 = 1 \mid H = 1) = 1 \cdot 0.98 = 0.98,$$

and marginalizing over H gives the probability that both tests are positive,

$$\begin{aligned} P(D_1 = 1, D_2 = 1) &= P(D_1 = 1, D_2 = 1 \mid H = 0)P(H = 0) + P(D_1 = 1, D_2 = 1 \mid H = 1)P(H \\ &= 0.0003 \cdot 0.9985 + 0.98 \cdot 0.0015 = 0.00176955. \end{aligned}$$

Bayes' theorem now gives

$$P(H = 1 \mid D_1 = 1, D_2 = 1) = \frac{P(D_1 = 1, D_2 = 1 \mid H = 1)P(H = 1)}{P(D_1 = 1, D_2 = 1)} = \frac{0.00147}{0.00176955} = 0.8307.$$

The second positive result raises the probability of infection from 13% to 83% (Figure 1.29). The second test is less accurate than the first, but it still helps, because its errors are independent of those of the first test. Without conditional independence, a second test can still help, but then we must model the joint distribution of the two results directly. At the other extreme, repeating a deterministic test reproduces the first result and adds nothing.

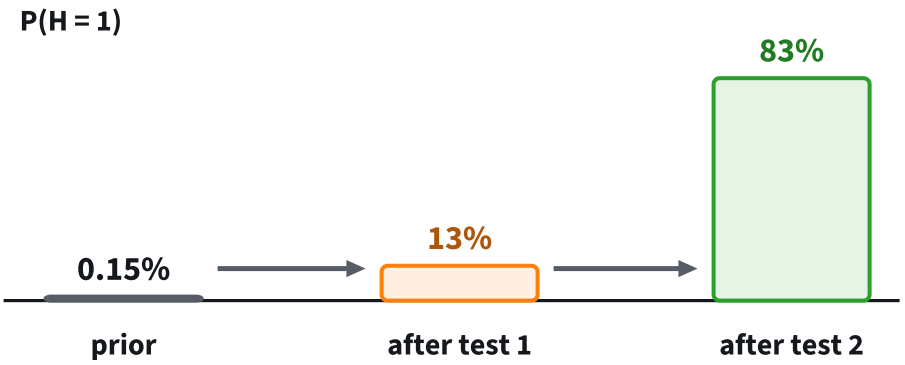


Figure 1.29: The probability that the patient has HIV: the prior, after one positive test, and after two conditionally independent positive tests.

Expectations can be conditioned like probabilities. The *conditional expectation*, or conditional mean, $\mathbb{E}[Y \mid X = x] = \sum_y y\, P(Y = y \mid X = x)$ is the average of Y over the outcomes in which $X = x$. In the HIV example the status H is an indicator, so $\mathbb{E}[H \mid D_1 = 1] = P(H = 1 \mid D_1 = 1) \approx 0.13$, and likewise $\mathbb{E}[H \mid D_1 = 1, D_2 = 1] \approx 0.83$.

Since we have a complete model of how the data arise, we can check the calculation by simulation. We draw the HIV status of two million patients, apply both tests to each patient, and compute the fraction of infected patients among those whose tests are both positive. Each draw compares a uniform random number with a probability, as in our first coin toss. The function `where` picks, for each patient, the probability that matches the HIV status, and indexing with the mask `D1 & D2` keeps the patients whose tests are both positive (Section 1.1).

```
n = 2000000
H = torch.rand(n) < 0.0015
D1 = torch.rand(n) < torch.where(H, 1.00, 0.01)
D2 = torch.rand(n) < torch.where(H, 0.98, 0.03)
H[D1 & D2].to(torch.float32).mean()  # Exact value: 0.8307
```

```
tensor(0.8220)
```

The simulated fraction lands close to the exact value, within about one percentage point. The agreement is looser than two million patients might suggest, because only the few patients with two positive tests enter the fraction. Of the two million simulated patients, only about $2{,}000{,}000 \cdot 0.00177 \approx 3{,}500$ test positive twice, and the simulated posterior is a frequency over these patients. Its standard error is $\sqrt{0.83 \cdot 0.17/3500} \approx 0.006$, so a deviation of one point is what we should expect. One more correct digit would take a hundred times as many patients.

Note that throughout this section the probabilities were handed to us: a fair coin, a test with known error rates. In learning nobody hands them to us, and a model with many parameters must estimate them from data. Chapter 2 begins with the simplest such model.

1.6.6 Summary

Probability describes how a random process generates data, and statistics reasons back from the data to the process. Probabilities obey three axioms, a joint distribution holds everything about several variables at once, and marginalizing sums over the values of a variable. Expectation and variance summarize a distribution, and since variances of independent variables add, the average of n independent draws has standard error $\sigma/\sqrt{n}$. A deviation of one standard error is what chance routinely produces, more than two are unusual, and many signal a real effect: 56 heads in 100 tosses are consistent with a fair coin, but 5,600 heads in $10{,}000$ tosses are strong evidence of bias. Conditioning restricts attention to the outcomes consistent with what we know and renormalizes, and Bayes' theorem reverses a conditional: a positive result from an accurate test for a rare disease leaves the disease unlikely (13%), and a second, conditionally independent positive raises it to 83%.

1.6.7 Exercises

1. **Union and intersection bounds.** Given two events with probabilities $P(\mathcal{A})$ and $P(\mathcal{B})$, derive the tightest upper and lower bounds on $P(\mathcal{A} \cup \mathcal{B})$ and $P(\mathcal{A} \cap \mathcal{B})$, and describe the configurations in Figure 1.23 for which each bound is attained.

2. **Portfolio return and risk.** Assume that you manage a portfolio of n stocks whose returns $\mathbf{s} \in \mathbb{R}^n$ are random with mean $\mu = \mathbb{E}[\mathbf{s}]$ and covariance matrix Σ. You invest a fraction $\alpha_i \geq 0$ of your capital in stock i, with $\sum_i \alpha_i = 1$.

    1. Compute the expected return and the variance of the return of the portfolio α.
    2. Assume that every stock has variance σ^2 and that every pair of distinct stocks has correlation $\rho \geq 0$. Compute the variance of the equally weighted portfolio $\alpha_i = 1/n$ and its limit as $n \to \infty$. Compare the case $\rho = 0$ with Equation 1.49.
    3. (Optional.) Formulate an optimization problem that maximizes the expected return while constraining the variance to an upper bound. This is the Markowitz portfolio problem (Mangram 2013). Methods for constrained problems of this kind are developed in Section 35.6.

3. code **Chebyshev for the coin-toss estimator.** Equation 1.49 gives the variance of the frequency $\hat{p}$ of heads after n tosses.

    1. For a fair coin and $n = 100$, use Chebyshev's inequality to bound $P(|\hat{p} - p| \geq 0.1)$.
    2. Estimate the same probability by simulating 10,000 repetitions of 100 tosses, compare it with the bound, and account for the gap.
    3. Determine how many tosses Chebyshev's inequality requires to guarantee $P(|\hat{p} - p| \geq 0.01) \leq 0.05$ for a fair coin, and compare with the n at which the standard error $\sqrt{p(1-p)/n}$ itself equals 0.01.

4. **Comparing two classifiers.** ⊠ Two classifiers are evaluated on the same test set of $n = 10{,}000$ images. Classifier A labels 9,050 images correctly and classifier B labels 9,000 correctly. Treat the test images as independent draws.

    1. Compute the standard error of each accuracy.
    2. Assume that the two accuracies are independent and compute the standard error of their difference. Is the difference of 0.005 large compared with the noise?
    3. The two accuracies are measured on the same images, so they are not independent. Suppose that the classifiers disagree on 300 images, A being correct on 175

of them and B on 125. For image i, let $Z_i = 1$ if only A is correct, $Z_i = -1$ if only B is correct, and $Z_i = 0$ otherwise, so that the difference in accuracy is the average of the Z_i. Compute the standard error of this average and compare it with your answer to the previous part.

4. Use $\mathrm{Var}(X - Y) = \mathrm{Var}(X) + \mathrm{Var}(Y) - 2\,\mathrm{Cov}(X, Y)$ to explain the difference. What sign does the covariance between the two classifiers' correctness on an image have, and why?

5. **Two kinds of uncertainty.** Give two examples of prediction tasks other than coin tossing. Choose the first so that more data can reduce the uncertainty about the outcome to any level, and the second so that it cannot. For each, state which part of the uncertainty is aleatoric and which is epistemic, and explain what sets the floor in the second example.

6. **Factorizing a chain.** Let X, Y, and Z be random variables such that Z is conditionally independent of X given Y, as in the chain of Figure 1.27.

   1. Write the joint probability $P(X, Y, Z)$ as a product of factors that each involve at most two of the variables.
   2. Generalize to a sequence $X_1, \ldots, X_T$ in which each X_t depends on the earlier variables only through X_{t-1}. How many numbers specify the joint distribution if every X_t takes k values, compared with an arbitrary joint distribution of T such variables?

7. **Conditioning on a common effect.** Toss two independent fair coins, and let $X, Y \in \{0, 1\}$ indicate heads on each. Let $Z = \max(X, Y)$ indicate that at least one coin came up heads. Compute $P(X = 1)$, $P(X = 1 \mid Z = 1)$, and $P(X = 1 \mid Z = 1, Y = 1)$. Use these values to decide whether X and Y are conditionally independent given $Z = 1$, and relate the change from the second value to the third to Figure 1.27.

8. **A second, correlated test.** ⊠ Revisit Section 1.6.5.4 with two new tests whose outcomes are conditionally independent for patients with HIV but not for healthy patients. Either test on its own has a false positive rate of 10% and a false negative rate of 1%:

$$P(D_i = 1 \mid H = 0) = 0.1, \quad P(D_i = 0 \mid H = 1) = 0.01 \quad \text{for } i = 1, 2. \tag{1.54}$$

   For $H = 1$, the joint distribution factorizes as $P(D_1, D_2 \mid H = 1) = P(D_1 \mid H = 1)\,P(D_2 \mid H = 1)$. For $H = 0$, the outcomes are coupled through $P(D_1 = D_2 = 1 \mid H = 0) = 0.02$. The baseline probability is $P(H = 1) = 0.0015$ as before.

   1. Work out the joint probability table for D_1 and D_2 given $H = 0$.
   2. Derive the probability that the patient has HIV after the first test returns positive, before the second test is run.
   3. Derive the probability that the patient has HIV after both tests return positive.
   4. Repeat the previous calculation for tests that are conditionally independent given $H = 0$ as well, and explain why the two posteriors differ.

9. **A defense lawyer's statistics.** A suspect had a documented history of abusing his now-murdered wife. His lawyer argues that, statistically, only one in a thousand wife-abusers goes on to murder his wife, so the history of abuse is weak evidence and the jury should acquit. Write M for "the suspect murdered his wife," K for "his wife was killed," and B for "he had a history of abusing her."

1. Express the lawyer's one-in-a-thousand figure and the posterior probability of guilt in these terms, and identify which probability the lawyer has confused with which.
2. Assume that, over the time span the lawyer's statistic refers to, an abused wife is murdered by someone other than her husband with probability 10^{-5}. Compute the posterior probability of guilt and compare it with the lawyer's figure.

Adapted from MIT 18.05, problem set 7, Problem 4.

1.7 Documentation

The sections of this chapter explained every function that they used. This courtesy cannot last. Every project soon needs a function that the book never mentions, or uses a familiar one in an unfamiliar way, and no book can keep up: the main module of a deep learning framework defines several hundred to a thousand public names, and new releases add names and deprecate others. What the book can teach instead is how to find out for ourselves, reliably, what a function does.

Reliably is the operative word, because a call can look harmless and still do something other than what we meant. Suppose that we need samples from a gamma distribution, a distribution of positive numbers often used for waiting times, with shape $k = 2$ and rate $\beta = 3$. Its mean is $k/\beta = 2/3$ and its variance $k/\beta^2 = 2/9$. Libraries disagree on how such a distribution is described: some take the rate β, others the scale $1/\beta$, and some neither. A call that passes the numbers 2 and 3 runs either way, but its samples have mean $2/3$ under one reading and $2 \cdot 3 = 6$ under the other. Nothing in the call tells us which reading applies. The function's signature and documentation do.

Instead of a list of functions, this section gives a procedure of four steps, shown in Figure 1.30. We discover the names that a library offers, inspect the arguments and the documentation of a candidate, read the reference pages or the source code when the documentation leaves a question open, and verify the behavior with a small example whose result we predicted. All four steps work inside a notebook, and by the end of the section they have drawn our gamma samples, with the right mean.

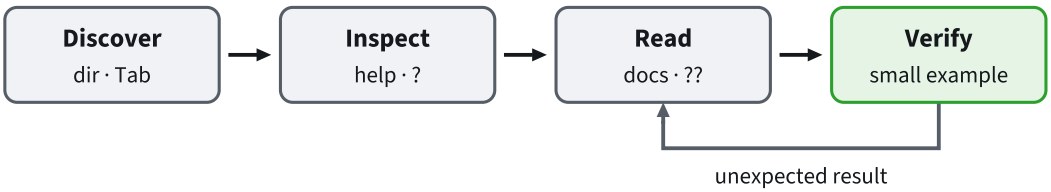


Figure 1.30: Four steps for finding out what a function does: discover the names that a module defines, inspect the signature and docstring of a candidate, read the reference documentation or the source code, and verify the behavior with a small example. A result that contradicts the prediction, or answers a question that the prediction left open, sends us back to read further.

Besides the framework, we import three helpers from Python's standard library: `pprint` prints long lists compactly, `inspect` reads the signature of a function, and `textwrap` wraps long lines.

```
import inspect
import textwrap
from pprint import pprint
import torch
```

1.7.1 Discovering Names

Our first question is whether the library can sample from a gamma distribution at all, and under which name. We know roughly where to look, in the module for random numbers, but not what its functions are called. The built-in function `dir` lists the names that a module defines. Names that begin with an underscore belong to the internals of the module, so we drop them and print the first twenty of the rest:

PyTorch keeps its probability distributions in the module `torch.distributions`.

```
pprint([name for name in dir(torch.distributions)
        if not name.startswith('_')][:20], compact=True)
```

```
['AbsTransform', 'AffineTransform', 'Bernoulli', 'Beta', 'Binomial',
 'CatTransform', 'Categorical', 'Cauchy', 'Chi2', 'ComposeTransform',
 'ContinuousBernoulli', 'CorrCholeskyTransform',
 'CumulativeDistributionTransform', 'Dirichlet', 'Distribution', 'ExpTransform',
 'Exponential', 'ExponentialFamily', 'FisherSnedecor', 'Gamma']
```

The names are distribution classes, such as `Bernoulli`, `Categorical`, and `Gamma`. Each class is created with the parameters of its distribution and then sampled from. Names that end in `Transform` are invertible maps, which `TransformedDistribution` applies to an existing distribution to build a new one.

A notebook offers the same list while we type: writing the name of a module followed by a dot and pressing Tab lists the names that the module defines, narrowed to those that match the letters typed next. When we do not know which module to look in, the search box of the framework's online reference searches all of them.

1.7.2 Reading Documentation

The list has an entry for the gamma distribution. To find out how to call it, the built-in function `help` prints its *docstring*, the documentation written into its code. A docstring says what the function does, what each argument means, and what the function returns, and it often ends with examples. The first lines show the function's *signature*, the list of arguments that it accepts. Three conventions of Python are all we need to read one. An argument written as `name=value` has a default value and may be omitted. Arguments that follow a lone `*` can be passed only by keyword. An argument written as `*name` collects any number of positional arguments; our sampler has none of these, but the exercises meet one.

The whole docstring is worth reading once in a notebook, but it is long, and our question is narrow: does the sampler take a rate or a scale? We therefore print the signature, which `inspect.signature` returns, and only those lines of the docstring that mention a rate or a scale. Since we print single lines, a sentence may arrive cut off; the whole of it is one `help` call away.

```
def doc_lines(f, *words):
    """Print the signature of f and the lines of its docstring with a word."""
    print(textwrap.fill(f'{f.__name__}{inspect.signature(f)}', 79,
                        subsequent_indent='    '))
    for line in inspect.getdoc(f).splitlines():
        if any(word in line for word in words):
            print('    ', line.strip())
```

```
doc_lines(torch.distributions.Gamma, 'rate', 'scale')
```

```
Gamma(concentration: torch.Tensor | float, rate: torch.Tensor | float,
    validate_args: bool | None = None) -> None
    Creates a Gamma distribution parameterized by shape :attr:`concentration` and :attr:`rate`.
    >>> m.sample()  # Gamma distributed with concentration=1 and rate=1
    rate (float or Tensor): rate parameter of the distribution
    (often referred to as beta), rate = 1 / scale
```

With the signature and these lines, we can predict what a call does before we run it:

`Gamma` is a class with the arguments `concentration`, which the docstring calls the shape parameter k, and `rate`, the rate β itself, which it also states as $1/\text{scale}$. One of the printed lines is an example from the end of the docstring, in which `m` is a `Gamma(1, 1)`; it confirms the vocabulary. We predict that samples of `Gamma(2.0, 3.0)` have mean $2/3$ and variance $2/9$.

In a notebook, two shortcuts show the same documentation. Appending `?` to a name shows the docstring in a separate pane, and appending `??` also shows the source code of the function. The source answers questions that a docstring leaves open, such as how an argument is converted or checked. Unfortunately, only functions written in Python have source code to show. Many core functions are compiled from C++, and for those `??` shows the docstring alone.

When the docstring is not enough, the framework's reference documentation describes every public function on a page of its own, and its tutorials show the functions in use.

For PyTorch, these are the API reference and the tutorials.

1.7.3 Verifying Behavior

Our reading of a docstring can be wrong, and occasionally the docstring itself is out of date. A small example settles the question, provided that we predict its result before we run it, so that the run could prove us wrong. A run that merely completes without an error proves little. Our prediction can fail: had we read the rate as a scale, the samples would have mean 6 instead of $2/3$. We draw 10,000 samples and compare their mean and variance with $2/3 \approx 0.667$ and $2/9 \approx 0.222$. The mean of 10,000 independent draws has standard error $\sqrt{(2/9)/10{,}000} \approx 0.005$ (Section 1.6), so the sample mean should lie within about 0.01 of $2/3$:

```
samples = torch.distributions.Gamma(2.0, 3.0).sample((10000,))  # k = 2, rate 3
samples.mean(), samples.var()
```

```
(tensor(0.6681), tensor(0.2284))
```

The prediction holds: the mean lies close to $2/3$ and the variance close to $2/9$, far from the values that the other reading would give. When a result contradicts the prediction, or answers a question that the prediction left open, we go around the loop again: we read further, in the reference documentation or the source, and test the new prediction.

Coding assistants answer many such questions directly. Asked for gamma samples with a given rate, an assistant proposes a call, and the call usually looks plausible. Such a suggestion is a candidate, and it goes through the same steps as a name we found ourselves: we check that the name exists, for example with `dir` or `hasattr`, read its signature, predict the result, and run a small example. The last step matters most, because a wrong suggestion need not fail: it can name a function that exists, fit its signature, and run, yet compute something else.

With this procedure the tools of the chapter are complete: tensors, tables, linear algebra, derivatives, probability, and a way to find what the book leaves out. Chapter 2 uses them to build and train a first model.

1.7.4 Summary

No text can record the details of every function in a deep learning framework, and a call that looks harmless may read its arguments differently than we assume: the same two numbers describe a gamma distribution by its rate in one library and by its scale in another. Four steps settle such questions from within a notebook. `dir` and Tab completion discover the names that a module defines, and `help` and `?` show the docstring of a candidate and its signature, which says what each argument means. The reference documentation and `??` supply the details that the docstring omits, and a small example whose result we predicted verifies our reading. Suggestions from coding assistants are candidates that go through the same steps.

1.7.5 Exercises

1. code **Two integers or one tuple.** `torch.ones` creates a tensor of ones. Read its signature with `help(torch.ones)`, which begins `ones(*size, *, ...)`, and predict, before running anything, what each of `torch.ones(2, 3)`, `torch.ones((2, 3))`, `torch.ones(size=(2, 3))`, and `torch.ones(shape=(2, 3))` returns or raises. Then run the four calls and compare. Name the convention of this section that decides each case.

2. code **Standard-deviation conventions.** For n numbers whose squared deviations from their mean sum to S, a library computes the standard deviation either as $\sqrt{S/n}$ or as $\sqrt{S/(n-1)}$. Section 1.2 standardized columns with the pandas method `std`. Compute the standard deviation of the values 1.0, 2.0, 3.0 and 4.0 with `pd.Series.std`, with `torch.std`, and with `numpy.std`. Use `help` on each function to find out how it sets the divisor, state the default divisor of each, and check the three results against a hand computation. Then make all three results agree.

3. code **Checking a coding assistant's answer.** Ask a coding assistant the two questions below. Check each function it names in three steps: `dir` or `hasattr` shows that the name exists, `help` or `?` shows that the signature fits the suggested call, and a tiny example shows the claimed behavior.

    1. Ask how to split a PyTorch tensor into three equal parts along an axis.

2. Ask for a single PyTorch function that reorders the axes of a tensor so that their lengths increase. Whatever the assistant answers, check each named function in the three steps, state which step, if any, it fails, and say whether the claimed behavior holds. If no suggestion passes, build the operation from functions that do, and test it on a tensor of shape $(4, 2, 3)$.

4. code **Reading the source.** The plotting function `d2l.plot`, defined in the online appendix "Utility Functions and Classes", accepts its data in several forms: a single tensor, a tensor of x-values with one tensor of y-values, or a tensor of x-values with several curves. Import `d2l` as at the start of this section, read the source of `d2l.plot` with `d2l.plot??` or your editor's go-to-definition, and find the lines that decide how many curves it draws and which values serve as their x-coordinates. Let `x` hold 50 evenly spaced values from -2 to 2 (use `linspace`), and let `Y` be a matrix of shape $(3, 50)$. Predict from the source what `d2l.plot(x)`, `d2l.plot(x, x ** 2)`, and `d2l.plot(x, Y)` draw, in particular whether `Y` gives three curves of 50 points or fifty curves of 3 points. Then run the three calls and compare.

5. code **Tests that can fail.** ☒ A verifying run is informative only if it would have given a different result had the claim been false.

   1. Find the framework's elementwise rounding function with `dir`, and design an input of at most four numbers on which three rules give three different results: round halves up, round halves away from zero, and round halves to the nearest even integer. Run the function on this input and state which rule the framework follows.
   2. A sort is *stable* if elements that compare equal keep their original order. Find the framework's `argsort`, and read what its docstring promises about stability. Then test stability on the input $(1, 0, 1, 0)$ and on 1,000 random integers from $\{0, 1\}$. Explain why a passing test on a short input cannot establish stability while a single failure refutes it, and state whether the behavior you observe agrees with the docstring.

Chapter 2

Linear Regression

Looking at a house for sale, most of us estimate its price by comparison. We recall what similar houses sold for, and adjust for a larger or older house. A learning method does the same with numbers, finding in past sales a rule that maps area, age and other measurements to a price. How should we find the rule, and how far can we trust it for a house that has not been sold?

Linear regression answers with the simplest rule: the prediction is a weighted sum of the measurements plus a constant, with weights that make the squared errors on past sales small. Each weight is one adjustment, such as the price per square meter. The method comes from astronomy. Legendre published it in 1805 to compute the orbits of comets. Gauss, whose orbit for the lost asteroid Ceres let astronomers find it again in 1801, published his account in 1809 and said he had used the method since 1795 (Legendre 1805; Gauss 1809). Simple as it is, linear regression is a complete learning method, with a model, a loss, an algorithm that improves the parameters, and predictions for examples the model has not seen.

Chapter 1 supplied the tools; this chapter assembles them into a learning procedure and asks what can go wrong. A model can fit its training data well and still predict new data badly, and an implementation can be wrong without raising an error. Neither failure announces itself, and both recur in every later chapter, so we study them here, where the model is simple enough to analyze.

The mathematics comes first and takes one section: Section 2.1 derives the model, its loss, and minibatch stochastic gradient descent, the algorithm that trains it. The next four sections turn it into code that we can trust, written first by hand and checked against data whose true parameters we know (Section 2.2, Section 2.3). We then package that code into a few classes that every later chapter reuses (Section 2.4), and replace our parts with those of a framework (Section 2.5). The last two sections return to the question of trust: when a model that fits its training data predicts new data well (Section 2.6), and how to limit its flexibility by degrees rather than by dropping features (Section 2.7).

The next chapter applies the same procedure to classification, and the one after replaces the linear model by a multilayer network.

Resources and Further Reading

The resources are grouped into books, courses, and papers, and each names the section it accompanies. All are freely accessible online except the two papers, as noted.

Books

- An Introduction to Statistical Learning (ISL) — James, Witten, Hastie & Tibshirani — free PDF (R and Python editions); the gentlest rigorous treatment of regression, classification, and regularization, with worked labs; companion to Section 2.1 and Section 2.6.
- The Elements of Statistical Learning (ESL) — Hastie, Tibshirani & Friedman — free PDF; the graduate companion to ISL, with deep coverage of shrinkage, the bias–variance tradeoff, and model selection (Section 2.6, Section 2.7).
- Pattern Recognition and Machine Learning — Christopher Bishop — free PDF from Microsoft Research; a probabilistic, Bayesian-flavoured account of linear models and regularization (Section 2.1, Section 2.7).
- Probabilistic Machine Learning: An Introduction — Kevin Murphy — free draft PDF; a modern, unifying probabilistic perspective spanning regression through deep learning (Section 2.1, Section 2.6).
- Understanding Deep Learning (UDL) — Simon Prince — free PDF; a figure-rich modern treatment whose loss-function chapter generalizes the "match the loss to the noise model" recipe of Section 2.1 into a full menu.
- Deep Learning: Foundations and Concepts — Bishop & Bishop (2024) — free to read online; the successor to PRML, connecting the probabilistic view of linear regression and regularization directly to deep networks (Section 2.1, Section 2.7).
- Introduction to Applied Linear Algebra: Vectors, Matrices, and Least Squares (VMLS) — Boyd & Vandenberghe — free PDF, slides, and code; an applications-first text built around least squares, the computation of Section 2.1.

Courses and video lectures

- Supervised Machine Learning: Regression and Classification — Andrew Ng (Stanford / DeepLearning.AI) — free to audit; the foundational course covering both settings of this part of the book, starting from linear regression trained by gradient descent (Section 2.3).
- Statistical Learning — Hastie & Tibshirani (Stanford Online, edX) — free to audit; lecture series following ISL, including regression, cross-validation, ridge, and the LASSO (Section 2.6, Section 2.7).

Foundational papers

- Ridge Regression: Biased Estimation for Nonorthogonal Problems — Hoerl & Kennard (1970), *Technometrics* — the origin of ℓ_2 regularization, the weight decay of Section 2.7 (paywalled).
- Regression Shrinkage and Selection via the Lasso — Tibshirani (1996), *JRSS-B* — introduces the ℓ_1-penalized counterpart, contrasting sparse selection with ridge's shrinkage (paywalled).

2.1 Linear Regression

Suppose you are about to buy a house with 150 square meters of living area, built 20 years ago, and want to know what a fair price is. You collect the recent sales in the neighborhood and note for each one its area, its age, and the price it fetched. Plotted against area, the prices scatter around a rising line: bigger houses cost more, not exactly but on average. If we can find that line, or with both area and age the corresponding plane, we can read off a price for the house in question and for any other house that has not been sold.

Predicting a number from measurements is called *regression*. It arises whenever the quantity of interest is numerical: the price of a home or a stock, a patient's length of stay in hospital, the demand for a product next quarter. Linear regression predicts the number as a weighted sum of the measurements. The method is two centuries old (Legendre 1805; Gauss 1809), and it is still the first thing to try on a regression problem. For us it has a second virtue. It is the simplest model in which every part of learning appears: a model with parameters, a loss that measures how badly the parameters fit the data, an algorithm that improves them, and predictions for new examples. The networks of later chapters replace the model and keep the rest.

This section builds these parts with the matrix–vector products, gradients, and expectations of Chapter 1. We choose the parameters that minimize the squared prediction error, in closed form, which has a simple geometric meaning, and by minibatch stochastic gradient descent, the algorithm that trains every network in this book. We then show that the squared error is the loss implied by Gaussian noise, which tells us when to use another loss.

```
import time
import numpy as np
import torch
```

2.1.1 The Model and the Loss

2.1.1.1 Features and Labels

Machine learning has names for the parts of this problem. The sales we collected form the *training set*, and each sale is an *example* (also called a data point, instance, or sample). The quantity we want to predict, the price, is the *label* (also target), and the quantities it is predicted from, area and age, are the *features* (also covariates). We write n for the number of examples, $\mathbf{x}^{(i)}$ for the features of example i, and $y^{(i)}$ for its label: superscripts in parentheses count examples and subscripts count features, so $x_j^{(i)}$ is feature j of example i. For the houses, $\mathbf{x}^{(i)} = (x_1^{(i)}, x_2^{(i)})$ holds the area and the age of sale i, and $y^{(i)}$ its price.

2.1.1.2 The Linear Model

A model states how the features determine the prediction. Linear regression assumes that the price is, on average, a weighted sum of the features plus a constant:

$$\text{price} = w_{\text{area}} \cdot \text{area} + w_{\text{age}} \cdot \text{age} + b. \tag{2.1}$$

The numbers w_{area} and w_{age} are the *weights*. Each says how much the price changes per unit of its feature. With $w_{\text{area}} = 3{,}000$ dollars per square meter, $w_{\text{age}} = -1{,}000$ dollars per year,

and $b = 50{,}000$ dollars, the house of 150 square meters built 20 years ago is predicted to cost $3{,}000 \cdot 150 - 1{,}000 \cdot 20 + 50{,}000 = 480{,}000$ dollars.

The constant b is the *bias* (also called offset or intercept), the prediction when all features are zero. No house has zero area, but without b every fit would be forced through the origin, which the data have no reason to respect. Strictly speaking, a linear function plus a constant is called affine, but the name linear regression is standard.

With d features, the prediction $\hat{y}$ is

$$\hat{y} = w_1 x_1 + \cdots + w_d x_d + b = \mathbf{w}^\top \mathbf{x} + b, \tag{2.2}$$

where the feature vector $\mathbf{x} = (x_1, \ldots, x_d)$ and the weight vector $\mathbf{w} = (w_1, \ldots, w_d)$ lie in $\mathbb{R}^d$. For all n examples at once, we stack the feature vectors as the rows of the *design matrix* $\mathbf{X} \in \mathbb{R}^{n \times d}$, with one row per example and one column per feature. The predictions $\hat{\mathbf{y}} \in \mathbb{R}^n$ are then a matrix–vector product,

$$\hat{\mathbf{y}} = \mathbf{X}\mathbf{w} + b, \tag{2.3}$$

in which the scalar b is added to every entry by broadcasting (Section 1.1.3.2), and the labels form the vector $\mathbf{y} = (y^{(1)}, \ldots, y^{(n)})$. A framework evaluates Equation 2.3 for a *minibatch*, a group of examples processed together, with one call to a matrix routine, which is far faster than a loop over the examples.

No real dataset satisfies $y^{(i)} = \mathbf{w}^\top \mathbf{x}^{(i)} + b$ exactly for every i. Two houses with the same area and age sell for different prices, because of features we did not record, such as the view, and because of chance. What, then, does the model claim? It describes the average label: we assume that the mean price of houses with features $\mathbf{x}$, which we write as the conditional expectation $\mathbb{E}[y \mid \mathbf{x}]$, equals $\mathbf{w}^\top \mathbf{x} + b$, and we treat the deviation of each label from it as noise. Our goal is to choose $\mathbf{w}$ and b so that predictions for new examples from the same population are accurate. For that we need two things: a measure of how good given parameters are, and a procedure that improves them.

2.1.1.3 The Squared Loss

A *loss function* measures the error of a prediction by a single number, zero for a perfect prediction and larger for worse ones. The most common loss for regression is the squared error. For example i, with prediction $\hat{y}^{(i)}$ and label $y^{(i)}$, it is

$$l^{(i)}(\mathbf{w}, b) = \frac{1}{2}\left(\hat{y}^{(i)} - y^{(i)}\right)^2. \tag{2.4}$$

The difference $r^{(i)} = \hat{y}^{(i)} - y^{(i)}$ between prediction and label is the *error* of example i, the vertical gap between a data point and the line in Figure 2.1 (statisticians call its negative, $y^{(i)} - \hat{y}^{(i)}$, the residual). The factor $\frac{1}{2}$ is there for convenience only: it cancels the 2 that differentiation produces, so that $\partial l^{(i)} / \partial \hat{y}^{(i)} = r^{(i)}$. It is a convention of this book and by no means a universal one. The mean squared error built into most libraries omits it, so their loss and its gradient are twice ours. The minimizer does not notice. Gradient descent does, as we will see, and whenever the two losses meet later in the book this factor of 2 is the whole difference between them.

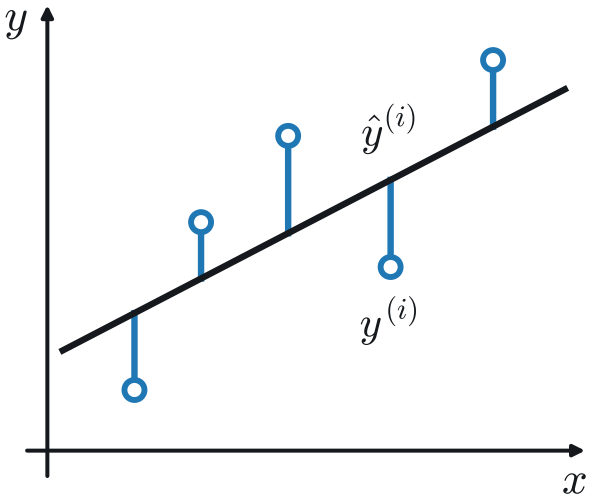


Figure 2.1: Fitting a line to one-dimensional data. The error of example i is the vertical gap between its prediction $\hat{y}^{(i)}$ on the line and its label $y^{(i)}$.

The quality of the parameters on the whole training set is the average loss,

$$L(\mathbf{w}, b) = \frac{1}{n}\sum_{i=1}^{n} l^{(i)}(\mathbf{w}, b) = \frac{1}{2n}\sum_{i=1}^{n} \left(\mathbf{w}^\top \mathbf{x}^{(i)} + b - y^{(i)}\right)^2,$$

half the mean squared error. Training means finding the parameters that minimize it,

$$\hat{\mathbf{w}}, \hat{b} = \operatorname*{argmin}_{\mathbf{w}, b} \; L(\mathbf{w}, b). \tag{2.5}$$

Like $\hat{y}$, the hats mark estimates computed from data. Since the training data are fixed, L is a function of the parameters alone. Up to the factor $\frac{1}{2n}$, it is the loss of the line model in Section 1.4, with the bias written separately.

Squaring makes large errors count far more than small ones. An error of 10 costs a hundred times as much as an error of 1, so a single bad label can dominate the fit. How badly? To find out, we place twenty points exactly on the line $y = 2x$, change one label to 10,000, as a typing error might, and fit a line through the origin, $\hat{y} = wx$, in two ways.

Setting the derivative of the squared loss $\sum_i (wx^{(i)} - y^{(i)})^2$ to zero gives its minimizer $w = \sum_i x^{(i)} y^{(i)} / \sum_i (x^{(i)})^2$. The *mean absolute error* $\frac{1}{n}\sum_i |wx^{(i)} - y^{(i)}|$ has no such formula. With a single parameter, however, we can simply evaluate it for a fine grid of slopes and keep the slope with the smallest error.

```
x = np.arange(1.0, 21.0)
y = 2 * x
y[5] = 10000                          # corrupt a single label
```

```
w_sq = (x * y).sum() / (x * x).sum()    # squared-loss minimizer
w_grid = np.linspace(0, 30, 3001)       # slopes 0, 0.01, ..., 30
mae = np.abs(w_grid[:, None] * x - y).mean(axis=1)
w_mae = w_grid[mae.argmin()]            # MAE minimizer on the grid
print(f'true w: 2.00, squared loss: {w_sq:.2f}, MAE: {w_mae:.2f}')
```

```
true w: 2.00, squared loss: 22.88, MAE: 2.00
```

One corrupted label in twenty pulls the squared-loss slope from 2 to about 23, while the absolute-error slope stays at 2. Under the squared loss, the single error of nearly 10,000 outweighs everything the other nineteen points contribute. Under the absolute error it counts only in proportion to its size, and the nineteen points on the line hold the slope in place. Which behavior is right? That depends on what we believe about the noise, the question of Section 2.1.3.

2.1.2 Minimizing the Loss

Linear regression is one of the few models whose loss can be minimized in closed form. We derive that solution first, because its geometry shows what least squares does, and then turn to the iterative method that every other model needs.

2.1.2.1 The Normal Equations

The squared loss is a quadratic function of the parameters, so its gradient is linear in them, and the points where the gradient vanishes solve a system of linear equations. To write the gradient compactly, we absorb the bias into the weights. Appending to every example a feature that always equals 1 gives the extended design matrix $\tilde{\mathbf{X}} = [\mathbf{X}, \mathbf{1}]$, with $d+1$ columns, and appending b to the weights gives $\tilde{\mathbf{w}} = (\mathbf{w}, b)$. Then $\tilde{\mathbf{X}}\tilde{\mathbf{w}} = \mathbf{X}\mathbf{w} + b$, and minimizing L is the same as minimizing $\|\tilde{\mathbf{X}}\tilde{\mathbf{w}} - \mathbf{y}\|^2$. In Section 1.4 we computed the gradient of this function, $2\tilde{\mathbf{X}}^\top(\tilde{\mathbf{X}}\tilde{\mathbf{w}} - \mathbf{y})$. Setting it to zero gives the *normal equations*,

$$\tilde{\mathbf{X}}^\top \tilde{\mathbf{X}}\, \tilde{\mathbf{w}} = \tilde{\mathbf{X}}^\top \mathbf{y}, \tag{2.6}$$

a system of $d + 1$ linear equations in the $d + 1$ unknowns, the d weights and the bias.

A zero gradient does not in general mark a minimum. Here it does, and the argument is short. Let $\hat{\tilde{\mathbf{w}}}$ solve the normal equations and compare the loss at any other point $\hat{\tilde{\mathbf{w}}} + \mathbf{v}$. Expanding the square gives

$$\|\tilde{\mathbf{X}}(\hat{\tilde{\mathbf{w}}} + \mathbf{v}) - \mathbf{y}\|^2 = \|\tilde{\mathbf{X}}\hat{\tilde{\mathbf{w}}} - \mathbf{y}\|^2 + 2\mathbf{v}^\top \tilde{\mathbf{X}}^\top(\tilde{\mathbf{X}}\hat{\tilde{\mathbf{w}}} - \mathbf{y}) + \|\tilde{\mathbf{X}}\mathbf{v}\|^2. \tag{2.7}$$

The middle term vanishes by the normal equations, and the last term is never negative, so no point has a smaller loss than $\hat{\tilde{\mathbf{w}}}$. The loss is strictly larger at $\hat{\tilde{\mathbf{w}}} + \mathbf{v}$ unless $\tilde{\mathbf{X}}\mathbf{v} = \mathbf{0}$, that is, unless moving the parameters by $\mathbf{v}$ leaves every prediction unchanged. When the columns of $\tilde{\mathbf{X}}$ are *linearly independent*, meaning that $\tilde{\mathbf{X}}\mathbf{v} = \mathbf{0}$ only for $\mathbf{v} = \mathbf{0}$, the minimizer is therefore unique. Independence fails when one feature is a combination of others, such as the area in square meters and the area in square feet. Then many weight vectors make the same predictions (exercise 8).

With independent columns the matrix $\tilde{\mathbf{X}}^\top\tilde{\mathbf{X}}$ is invertible, and multiplying the normal equations by its inverse gives the minimizer

$$\hat{\tilde{\mathbf{w}}} = (\tilde{\mathbf{X}}^\top\tilde{\mathbf{X}})^{-1}\tilde{\mathbf{X}}^\top\mathbf{y}.$$

This formula is for analysis, not for computation. Forming $\tilde{\mathbf{X}}^\top\tilde{\mathbf{X}}$ amplifies rounding errors (Section 33.5.3.2), and computing an inverse is slower and less accurate than solving the system (Golub and Van Loan 1996). Libraries provide a least-squares solver, `lstsq`, which works directly with a factorization of $\tilde{\mathbf{X}}$.

Let us try this on the three-example line model of Section 1.4: inputs 0, 1, and 2, labels 1, 1, and 0, and the prediction $w_1 + w_2 \cdot \text{input}$. Its bias is w_1, so the column of ones comes first and the rows of $\tilde{\mathbf{X}}$ are $(1, \text{input})$. Then

$$\tilde{\mathbf{X}} = \begin{bmatrix} 1 & 0 \\ 1 & 1 \\ 1 & 2 \end{bmatrix}, \quad \tilde{\mathbf{X}}^\top\tilde{\mathbf{X}} = \begin{bmatrix} 3 & 3 \\ 3 & 5 \end{bmatrix}, \quad \tilde{\mathbf{X}}^\top\mathbf{y} = \begin{bmatrix} 2 \\ 1 \end{bmatrix}, \tag{2.8}$$

and the normal equations $3w_1 + 3w_2 = 2$ and $3w_1 + 5w_2 = 1$ give $\hat{w}_2 = -1/2$ and $\hat{w}_1 = 7/6$, the minimum that Section 1.4 located at $(7/6, -1/2)$. We let the library solve the same problem (the code calls $\tilde{\mathbf{X}}$ simply `X`) and print the weights, the errors $\mathbf{r} = \tilde{\mathbf{X}}\hat{\tilde{\mathbf{w}}} - \mathbf{y}$, and the largest entry of $\tilde{\mathbf{X}}^\top\mathbf{r}$ in absolute value, which the normal equations say is zero.

```
X = torch.tensor([[1.0, 0.0], [1.0, 1.0], [1.0, 2.0]])  # rows (1, input)
y = torch.tensor([[1.0], [1.0], [0.0]])
w_hat = torch.linalg.lstsq(X, y).solution
r = X @ w_hat - y                                        # errors
print('w_hat:', tuple(round(float(v), 4) for v in w_hat[:, 0]))
print('r:', tuple(round(float(v), 4) for v in r[:, 0]))
print(f'max |X^T r|: {float((X.T @ r).abs().max()):.0e}')
```

```
w_hat: (1.1667, -0.5)
r: (0.1667, -0.3333, 0.1667)
max |X^T r|: 2e-07
```

The library returns $\hat{\tilde{\mathbf{w}}} = (7/6, -1/2)$ and the errors $\mathbf{r} = (1/6, -1/3, 1/6)$, printed to four decimals. The vector $\tilde{\mathbf{X}}^\top\mathbf{r}$ is zero up to rounding errors of order 10^{-7}, the precision of float32.

2.1.2.2 Least Squares as Projection

The normal equations have a geometric meaning, which also explains their name. In Section 1.3 we saw that a matrix–vector product is a combination of the columns of the matrix, here with the entries of $\tilde{\mathbf{w}}$ as the coefficients. Each column holds one feature for all n examples, so it is a vector in $\mathbb{R}^n$, like the labels $\mathbf{y}$. As $\tilde{\mathbf{w}}$ varies, the predictions $\tilde{\mathbf{X}}\tilde{\mathbf{w}}$ range over all combinations of the columns, a set called the *column space* of $\tilde{\mathbf{X}}$. For two columns in $\mathbb{R}^3$ it is a plane through the origin. Minimizing $\|\tilde{\mathbf{X}}\tilde{\mathbf{w}} - \mathbf{y}\|$ asks for the point of the column space closest to the label vector $\mathbf{y}$.

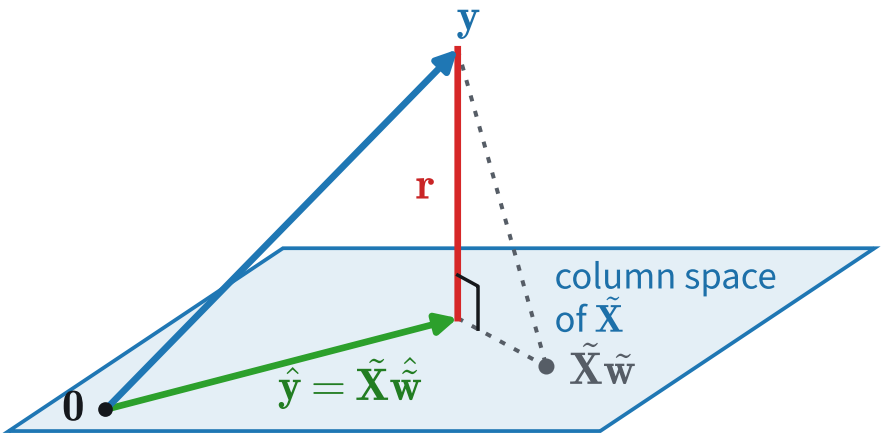


Figure 2.2: Least squares as projection. The predictions $\tilde{\mathbf{X}}\tilde{\mathbf{w}}$ range over the column space of $\tilde{\mathbf{X}}$. The fitted values $\hat{\mathbf{y}} = \tilde{\mathbf{X}}\hat{\tilde{\mathbf{w}}}$ are the foot of the perpendicular from the label vector $\mathbf{y}$, so the error vector $\mathbf{r} = \hat{\mathbf{y}} - \mathbf{y}$ meets the column space at a right angle. Any other prediction $\tilde{\mathbf{X}}\tilde{\mathbf{w}}$ lies farther from $\mathbf{y}$ (dotted): its squared distance adds the squared length of the dotted leg in the plane.

The closest point is the foot of the perpendicular from $\mathbf{y}$ (Figure 2.2, drawn for any matrix $\tilde{\mathbf{X}}$). The normal equations express this condition. Written as $\tilde{\mathbf{X}}^\top \mathbf{r} = \mathbf{0}$ with the errors $\mathbf{r} = \tilde{\mathbf{X}}\hat{\tilde{\mathbf{w}}} - \mathbf{y}$, they state that the error vector has zero dot product with every column of $\tilde{\mathbf{X}}$, so it is perpendicular to the column space; "normal" is an old word for perpendicular. The identity Equation 2.7 is the theorem of Pythagoras for the right triangle in the figure: for any other prediction $\tilde{\mathbf{X}}\tilde{\mathbf{w}}$, the squared distance to $\mathbf{y}$ is the squared length of the error vector plus the squared distance $\|\tilde{\mathbf{X}}\tilde{\mathbf{w}} - \hat{\mathbf{y}}\|^2$ within the column space. The fitted values $\hat{\mathbf{y}} = \tilde{\mathbf{X}}\hat{\tilde{\mathbf{w}}}$ are the *orthogonal projection* of $\mathbf{y}$ onto the column space (Section 33.1 treats projections in general).

In the three-example fit, the error vector $(1/6, -1/3, 1/6)$ is perpendicular to both columns: its dot product with the column of ones $(1, 1, 1)$ is $1/6 - 1/3 + 1/6 = 0$, and its dot product with the inputs $(0, 1, 2)$ is $-1/3 + 2/6 = 0$. Perpendicularity to the column of ones means that the errors sum to zero. This holds for every least-squares fit with a bias: on average the line neither overestimates nor underestimates the labels (exercise 2).

2.1.2.3 Minibatch Stochastic Gradient Descent

Most models in this book have no closed-form minimizer. They are trained by gradient descent, which Section 1.4 introduced: starting from initial parameters, it repeatedly takes a small step against the gradient of the loss, $\mathbf{w} \leftarrow \mathbf{w} - \eta \nabla_{\mathbf{w}} L(\mathbf{w})$, where the learning rate $\eta > 0$ sets the size of the step. Our loss is an average over all n examples, so with millions of examples every step takes a full pass over the data. This is wasteful: a small fraction of the examples would already show roughly which way to go. The opposite extreme uses a single randomly chosen example per step, which is called *stochastic gradient descent* (SGD) (Bottou 2010). Its steps are cheap but noisy, and they waste the hardware instead: a processor computes a matrix product over many examples much faster than the same arithmetic

one example at a time, as we measure below.

The compromise is a minibatch: at step t we draw a random set $\mathcal{B}_t$ of $|\mathcal{B}|$ examples and average the gradient over it (Li, Zhang, et al. 2014). Because the minibatch is drawn at random, its average gradient equals the full gradient in expectation; in the language of Section 1.6, it is an unbiased estimate. Its variance falls in proportion to $1/|\mathcal{B}|$, since the variance of an average of independent draws is the variance of one draw divided by their number. Written for the weights and the bias separately, the update of *minibatch stochastic gradient descent* is

$$(\mathbf{w}, b) \leftarrow (\mathbf{w}, b) - \frac{\eta}{|\mathcal{B}|} \sum_{i \in \mathcal{B}_t} \nabla_{(\mathbf{w},b)}\, l^{(i)}(\mathbf{w}, b). \tag{2.9}$$

For the squared loss the chain rule gives the gradients explicitly: the derivative of $l^{(i)}$ with respect to the prediction is the error $r^{(i)} = \hat{y}^{(i)} - y^{(i)}$, and the derivative of the prediction is $\mathbf{x}^{(i)}$ with respect to $\mathbf{w}$ and 1 with respect to b. The updates are therefore

$$\begin{aligned} \mathbf{w} &\leftarrow \mathbf{w} - \frac{\eta}{|\mathcal{B}|} \sum_{i \in \mathcal{B}_t} \mathbf{x}^{(i)} \left(\mathbf{w}^\top \mathbf{x}^{(i)} + b - y^{(i)}\right), \\ b &\leftarrow b - \frac{\eta}{|\mathcal{B}|} \sum_{i \in \mathcal{B}_t} \left(\mathbf{w}^\top \mathbf{x}^{(i)} + b - y^{(i)}\right). \end{aligned} \tag{2.10}$$

Each example pulls the weights along its own feature vector, in proportion to its error, and the bias moves by the average error. Training initializes $\mathbf{w}$ and b, often at random, and repeats the update for a fixed number of steps or until the loss stops improving. In practice the name SGD refers to this minibatch version: the optimizers that frameworks call SGD implement it, and so does this book from here on.

The minibatch size $|\mathcal{B}|$ and the learning rate η are chosen by us rather than learned; such settings are called *hyperparameters*. Powers of 2 between 32 and 256 are common starting points for the minibatch size, limited by memory and by how many examples the hardware processes in parallel. The last sections of this chapter show how to choose hyperparameters with data held out from training.

The learning rate is tied to the scale of the loss. Multiplying the loss by a constant $c > 0$ multiplies its gradient by c, so SGD takes the same steps on cL with learning rate η as on L with learning rate $c\eta$. A learning rate therefore has a meaning only together with the loss it is used with. A library loss without our factor $\frac{1}{2}$ needs half our learning rate to take the same steps, and a loss summed over the minibatch instead of averaged needs it divided by $|\mathcal{B}|$.

2.1.2.4 Gradient Noise and the Learning Rate

Minibatch SGD with a constant learning rate does not converge to the minimizer, even on least squares. At the minimizer the gradient of the full loss vanishes, but the gradient on a minibatch does not: the terms $\mathbf{x}^{(i)} r^{(i)}$ cancel in the sum over the whole training set, which is what $\tilde{\mathbf{X}}^\top \mathbf{r} = \mathbf{0}$ states, but not within a minibatch. The parameters therefore keep moving and fluctuate in a cloud around the minimizer, the *noise ball* (Figure 2.3). Full-batch gradient descent has no such noise and converges.

How large is the cloud? A single weight is enough to see it. Let $u = w - \hat{w}$ be its distance from the minimizer, and scale the feature so that the gradient of the full loss is u. A step of SGD then replaces u by $(1-\eta)u - \eta\xi$, where the noise ξ of the minibatch gradient has mean zero and variance $s^2/|\mathcal{B}|$, with s^2 the variance of the gradient of a single example. Near the minimizer ξ is nearly independent of u, so the variance v of u becomes $(1-\eta)^2 v + \eta^2 s^2/|\mathcal{B}|$, and the cloud has settled when this equals v again: at $v \approx \eta s^2/(2|\mathcal{B}|)$ for small η. The mean squared distance from the minimizer is proportional to $\eta/|\mathcal{B}|$, and it grows with the label noise, which enters through s^2.

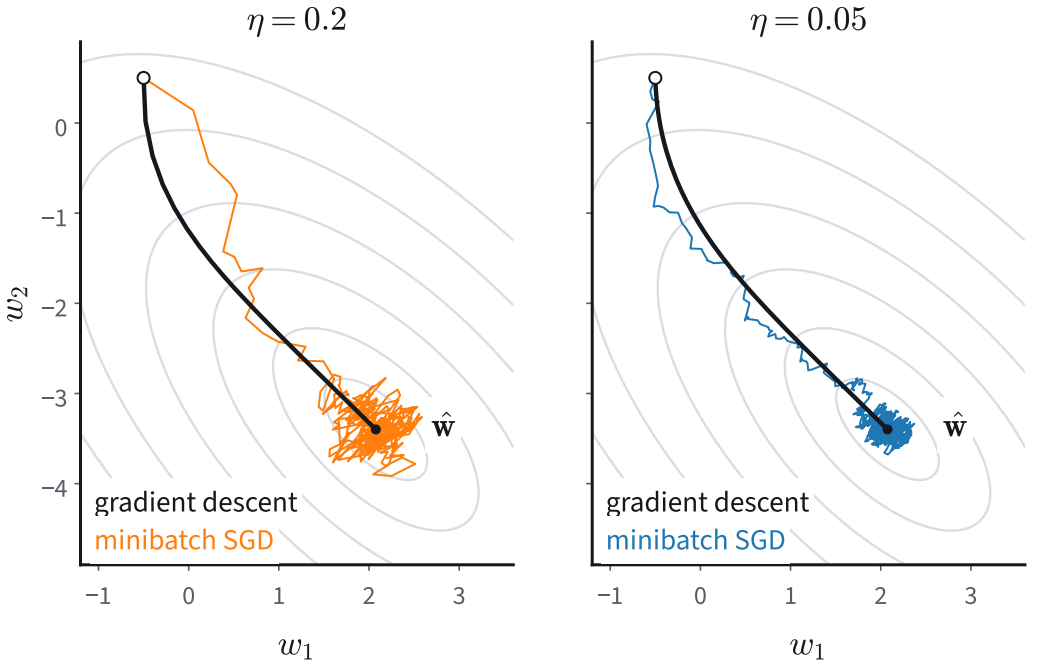


Figure 2.3: Gradient descent (black) and minibatch SGD with minibatches of 10 (color) on a two-parameter least-squares problem with noisy labels, from the same starting point (open circle). Gray ellipses are contours of the loss. Gradient descent converges to the minimizer $\hat{\mathbf{w}}$. SGD follows the same route on average but ends in a cloud of fluctuations around $\hat{\mathbf{w}}$, and quartering the learning rate (right) halves the width of the cloud.

Quartering η quarters v and so halves the width of the noise ball, as the figure shows. Unfortunately, each step then also shrinks the distance less, and SGD needs about four times as many steps to get close. In practice the learning rate is therefore lowered during training, following a *schedule*; the appendix derives the size of the cloud for several weights (Section 35.4.1). Note also that two runs from the same start end at different points of the cloud, since their minibatches differ. The trained parameters are themselves random, which is why the results quoted in this book are ranges rather than single numbers.

2.1.2.5 Vectorization for Speed

We have claimed twice that one matrix product over many examples beats a loop over them. It is time to measure. We predict the labels of 1,000 examples with the weights $\mathbf{w} = (2, -3.4)$ and the bias $b = 4.2$, once in a Python loop over the examples and once

with a single call into compiled code (Section 1.1). With $\mathbf{w}$ stored as a 2×1 matrix, the product `X @ w + b` serves both for one example, a 1×2 row of `X`, and for all of them.

```
n = 1000
X = torch.ones(n, 2)               # the values do not affect the timing
w = torch.tensor([[2.0], [-3.4]])
b = 4.2
# Wait until the device has finished (CUDA runs asynchronously)
wait = lambda x: torch.cuda.synchronize() if torch.cuda.is_available() else None
```

The function `seconds` times one call of a function, after a first call that absorbs one-time setup costs. Before it stops the clock it calls `wait`, defined above, because a framework may return from a call while an accelerator is still at work.

```
def seconds(f):
    """Time one call of f, after a warm-up call."""
    wait(f())
    start = time.perf_counter()
    wait(f())
    return time.perf_counter() - start
```

```
loop = seconds(lambda: [X[i:i + 1] @ w + b for i in range(n)])
vectorized = seconds(lambda: X @ w + b)
print(f'one example at a time: {loop * 1e3:8.3f} ms')
print(f'all examples at once:  {vectorized * 1e3:8.3f} ms')
print(f'speedup: {loop / vectorized:.0f}x')
```

```
one example at a time:   11.358 ms
all examples at once:     0.024 ms
speedup: 482x
```

Computing all predictions at once is faster by a factor of hundreds or more, although the arithmetic is the same. The loop incurs the overhead of the Python interpreter and of launching a framework operation once per example; the matrix product incurs it once. The same holds for the gradient, which is why we train on minibatches rather than on single examples. After training, the same product computes the *prediction* $\hat{\mathbf{w}}^\top \mathbf{x} + \hat{b}$ for new examples, a step that practitioners also call inference.

2.1.3 Squared Loss and Gaussian Noise

So far the squared loss has been a convenient choice, smooth and with a closed-form minimizer, but too sensitive to the corrupted label. When is it the right choice? A probabilistic view gives the answer. We model how the labels scatter around the line and choose the parameters under which the observed labels are most probable.

2.1.3.1 The Normal Distribution

The *normal distribution*, or Gaussian distribution, with mean μ and variance σ^2 has the density

$$p(x) = \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left(-\frac{(x-\mu)^2}{2\sigma^2}\right).$$

The density is a bell-shaped curve, symmetric about μ, whose width is set by the standard deviation σ: about 68% of the probability lies within σ of the mean and about 95% within 2σ. Since the squared distance from the mean appears in the exponent, large deviations are extremely improbable: a value 5σ from the mean has density $e^{-12.5} \approx 4 \times 10^{-6}$ times that at the mean. We write $\epsilon \sim \mathcal{N}(\mu, \sigma^2)$ for a random variable with this distribution. Sums of many small independent effects are approximately normal (Section 36.3.2.3), which makes it the natural model for measurement noise.

Linear regression with Gaussian noise assumes that each label is the linear prediction plus an independent normal deviation,

$$y = \mathbf{w}^\top \mathbf{x} + b + \epsilon, \quad \epsilon \sim \mathcal{N}(0, \sigma^2). \tag{2.11}$$

The conditional mean $\mathbb{E}[y \mid \mathbf{x}]$ is then $\mathbf{w}^\top \mathbf{x} + b$, the assumption of Section 2.1.1.2, and the labels scatter around it with the same standard deviation σ for every $\mathbf{x}$ (Figure 2.4).

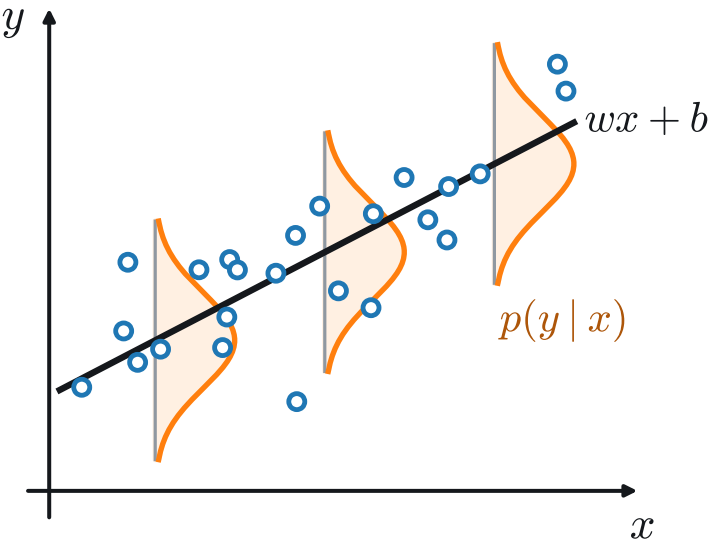


Figure 2.4: The Gaussian noise model. For every input x, the label y is drawn from a normal density $p(y \mid x)$ centered on the line $wx + b$, with the same width everywhere.

2.1.3.2 Maximum Likelihood

The noise model lets us judge parameters without choosing a loss in advance, because it says how probable the observed labels are under each choice of the parameters. Given parameters $\mathbf{w}$ and b, the density of a label y for features $\mathbf{x}$ is the normal density centered on the prediction:

$$p(y \mid \mathbf{x}) = \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left(-\frac{(y - \mathbf{w}^\top \mathbf{x} - b)^2}{2\sigma^2}\right). \tag{2.12}$$

A label close to the line has a high density and a label far from it a low one. Viewed as a function of the parameters, the density of the observed data is the *likelihood*, the quantity $P(D \mid H)$ of Bayes' theorem in Section 1.6 with the parameters in the role of the hypothesis. The *principle of maximum likelihood* chooses the parameters under which the observed labels are most probable. Since the examples are drawn independently, the density of all labels is the product

$$p(\mathbf{y} \mid \mathbf{X}) = \prod_{i=1}^{n} p(y^{(i)} \mid \mathbf{x}^{(i)}). \tag{2.13}$$

For the three examples of Section 2.1.2.1 and $\sigma = 1$, the fitted line $(7/6, -1/2)$ gives the labels a joint density of about 0.058, and the horizontal line through their mean one of about 0.045. Maximum likelihood prefers the first, and no other line does better.

A product of many small numbers is awkward to maximize, and in floating point it underflows to zero. The logarithm turns the product into a sum, and since it is increasing it does not move the maximizer. By convention we minimize instead of maximize, so we minimize the *negative log-likelihood*

$$-\log p(\mathbf{y} \mid \mathbf{X}) = \sum_{i=1}^{n} \left[\frac{1}{2}\log(2\pi\sigma^2) + \frac{1}{2\sigma^2}\left(y^{(i)} - \mathbf{w}^\top \mathbf{x}^{(i)} - b\right)^2\right]. \tag{2.14}$$

The first term does not depend on $\mathbf{w}$ or b, and the second is the sum of squared errors divided by $2\sigma^2$, so the minimizer over $\mathbf{w}$ and b is the least-squares solution for every σ. Least squares, it turns out, is maximum likelihood estimation under Gaussian noise. The appendix treats maximum likelihood in general (Section 36.6.2.3).

The derivation also explains the corrupted label. Under Gaussian noise with $\sigma = 1$, a label of 10,000 where the line predicts 12 lies about 10^4 standard deviations away, which has essentially zero probability. To make the data less improbable, maximum likelihood tilts the line toward that label, at the expense of the nineteen other points, each of which then costs only a moderate amount.

2.1.3.3 Choosing a Loss from a Noise Model

The same derivation applies to any noise model. We choose a conditional distribution $p(y \mid \mathbf{x})$ that describes how the labels scatter and minimize its negative log-likelihood. Gaussian noise yields the squared error, and other assumptions yield other losses (in the last row, $\nu(\mathbf{x})$ is the predicted rate of the count):

Output type	Noise model	Loss (negative log-likelihood)
real-valued, well-behaved noise	Gaussian	squared error (this section)

Output type	Noise model	Loss (negative log-likelihood)
real-valued, outliers or heavy tails	Laplace	absolute error (exercise 3)
positive, multiplicative fluctuations (prices)	Gaussian on $\log y$	squared error on $\log y$ (exercise 4)
counts (sales, arrivals)	Poisson	$\nu(\mathbf{x}) - y \log \nu(\mathbf{x})$ (exercise 5)

The Laplace distribution has the density $p(\epsilon) = \frac{1}{2\beta} \exp(-|\epsilon|/\beta)$ with scale $\beta > 0$. Its exponent falls off with $|\epsilon|$ instead of ϵ^2, so its tails carry far more probability than those of a Gaussian with the same variance (Figure 2.5, left), and its negative log-likelihood penalizes a large error only linearly. This is the robustness we observed: the Laplace model treats a label far from the line as unusual but possible, and does not tilt the line toward it. The figure also shows the *Huber loss*, quadratic for small errors and linear beyond a threshold δ, for data in which most of the noise is Gaussian but a few labels are corrupted. The next chapter uses the same principle to derive the loss for classification.

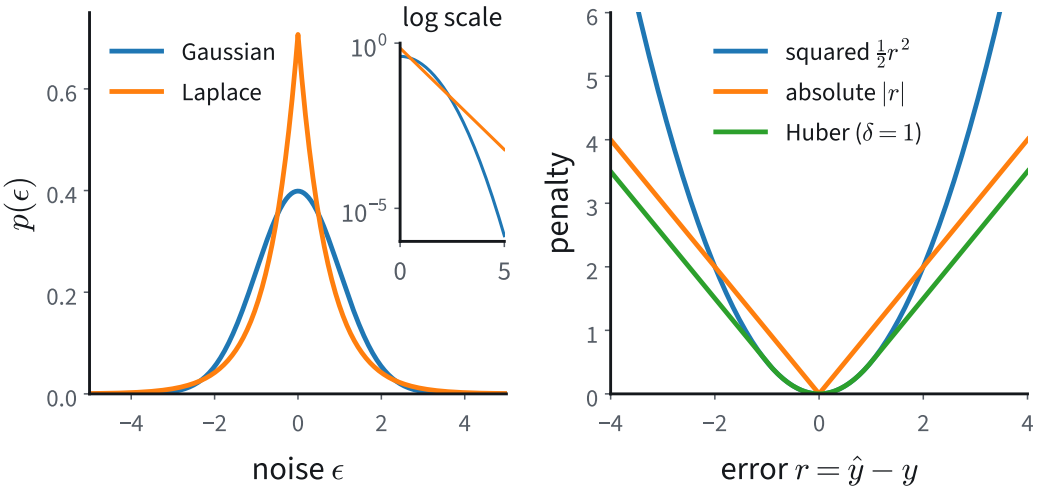


Figure 2.5: Matching the loss to the noise model. Left: Gaussian and Laplace densities with equal variance; the log-scale inset shows the Laplace tails carrying far more probability. Right: the penalty on an error that each noise model induces, together with the Huber loss between them.

2.1.4 Linear Regression as a Neural Network

Linear regression is also the smallest neural network. Figure 2.6 draws it as one. Each feature is an input unit, and a single output unit computes the weighted sum of the inputs plus the bias, $o_1 = \mathbf{w}^\top \mathbf{x} + b$, with one weight per connection. The inputs are given and only the output is computed, so the network has a single layer of computation. Since every input is connected to the output, the layer is called *fully connected* (or dense).

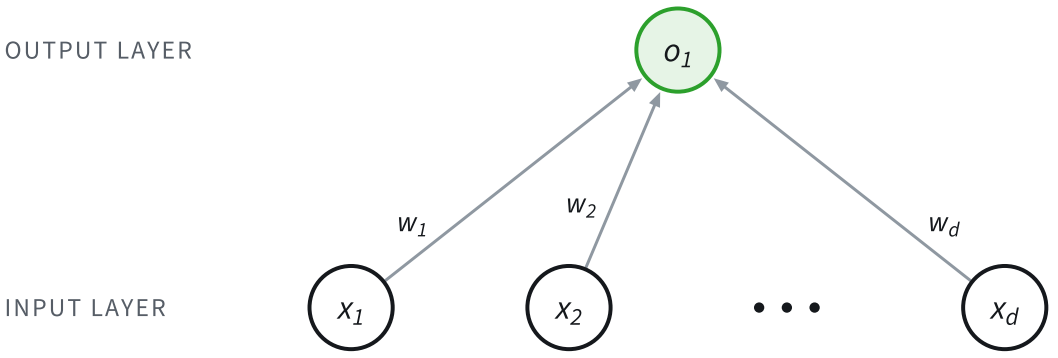


Figure 2.6: Linear regression as a network with one layer. The inputs $x_1, \dots, x_d$ feed a single output $o_1 = \mathbf{w}^\top \mathbf{x} + b$, with one weight w_j per connection.

The vocabulary of units and weighted connections comes from early models of biological neurons (McCulloch and Pitts 1943), but the analogy is loose. What we keep is the view of linear regression as a layer: the networks of later chapters stack many such layers, with nonlinear functions between them, and train them with the same kind of loss and the same minibatch SGD.

Two things are still missing. We have not trained anything yet, and training needs data, a model, and a loop in code; the next sections build them, starting with synthetic data whose right answer we choose ourselves. And a small training loss does not by itself guarantee accurate prices for houses that were not in the training set. The house of 150 square meters waits for real data, which arrive in Section 4.7.

2.1.5 Summary

Linear regression predicts a number as a weighted sum of features plus a bias, $\hat{y} = \mathbf{w}^\top \mathbf{x} + b$. The parameters that minimize the average squared error on the training set solve the normal equations, and the fitted values are the orthogonal projection of the label vector onto the column space of the design matrix. Minibatch stochastic gradient descent approaches the same minimizer with cheap, noisy steps and, at a constant learning rate, ends in a noise ball around it whose size grows with the learning rate. The squared error is the negative log-likelihood of Gaussian noise, which is why it is sensitive to outliers, and other noise models give other losses. As a network, linear regression is a single fully connected layer, the building block of the deep networks to come.

2.1.6 Exercises

1. **Fitting a constant.** Consider a model without features that predicts the same value b for every example. Fit it to labels $y^{(1)}, \dots, y^{(n)} \in \mathbb{R}$ by minimizing $\sum_i (y^{(i)} - b)^2$.
    1. Find an analytic solution for the optimal value of b.
    2. Fit a normal distribution $\mathcal{N}(b, \sigma^2)$ to $y^{(1)}, \dots, y^{(n)}$ by maximum likelihood in both b and σ^2. Show that the estimate of b is the solution of part (a) and find the estimate of σ^2.
    3. Change the loss to $\sum_i |y^{(i)} - b|$ and find all of its minimizers. Then replace a single label by an arbitrary value and determine how far the minimizers of each

of the two losses can move.

2. **Intercept and centering.** Appending a constant feature 1 to every input turns the affine model $\mathbf{w}^\top \mathbf{x} + b$ into a linear model in $(\mathbf{x}, 1)$. Centering the data removes the bias in a different way. Let $\bar{\mathbf{x}}$ and $\bar{y}$ be the averages of the features and of the labels over the training set.
    1. Show that every minimizer $(\widehat{\mathbf{w}}, \widehat{b})$ of the squared loss satisfies $\widehat{b} = \bar{y} - \widehat{\mathbf{w}}^\top \bar{\mathbf{x}}$, and conclude that the errors $\widehat{y}^{(i)} - y^{(i)}$ of a least-squares fit with a bias sum to zero.
    2. Show that $\widehat{\mathbf{w}}$ is a least-squares fit without bias to the centered data $(\mathbf{x}^{(i)} - \bar{\mathbf{x}},\ y^{(i)} - \bar{y})$.
3. **Laplace noise.** Assume that the additive noise ϵ follows the Laplace distribution with scale 1 (see Section 36.3), that is, $p(\epsilon) = \frac{1}{2} \exp(-|\epsilon|)$.
    1. Write out the negative log-likelihood of the data under the model $-\log p(\mathbf{y} \mid \mathbf{X})$.
    2. Determine whether a closed-form solution exists.
    3. Write the minibatch stochastic gradient descent update for $(\mathbf{w}, b)$ under this loss in the form of Equation 2.10, using 0 where $|\cdot|$ has no derivative, at zero, and elsewhere its slope $\text{sign}(\cdot) = \pm 1$. A substitute of this kind is called a *subgradient*. Compare how the size of each example's contribution depends on its error under this loss and under the squared loss.
    4. code Apply this update to the corrupted-label data of Section 2.1.1.3, for a line through the origin, using the full dataset in every step, the learning rate 0.002, and 2,000 steps from $w = 0$. The minimizer of the absolute error is exactly $w = 2$. Where do the iterates end? Explain why they do not settle at the minimizer, and propose a fix.
4. **Beyond Gaussian noise.** Suppose you use regression to estimate the price of houses or of stocks.
    1. Model the price directly as $y = \mathbf{w}^\top \mathbf{x} + b + \epsilon$ with $\epsilon \sim \mathcal{N}(0, \sigma^2)$, and take $\sigma = 100{,}000$ dollars. Compute the probability that the model assigns to a negative price for a house whose predicted price is $50{,}000$ dollars. Explain why no single σ suits both this house and a house whose predicted price is $5{,}000{,}000$ dollars.
    2. Now regress on the logarithm of the price, $\log \text{price} = \mathbf{w}^\top \mathbf{x} + b + \epsilon$, where $\epsilon \sim \mathcal{N}(0, \sigma^2)$ now acts on the log scale. Compute the conditional mean and standard deviation of the price and show that their ratio does not depend on $\mathbf{x}$. Determine whether $\exp(\mathbf{w}^\top \mathbf{x} + b)$ is an unbiased prediction of the price. The same modeling choice underlies the Black–Scholes model of option pricing (Black and Scholes 1973).
5. **Counting apples.** Suppose we want to use regression to estimate the *number* of apples sold in a grocery store.
    1. Identify the problems with a Gaussian additive noise model for such a target.
    2. The Poisson distribution models counts. It is given by $p(k \mid \nu) = \nu^k e^{-\nu} / k!$, where $\nu > 0$ is the rate and $k \in \{0, 1, 2, \ldots\}$ is the observed count. Prove that $\mathbb{E}[k] = \nu$.
    3. Derive the negative log-likelihood of a count k under a rate $\nu(\mathbf{x})$ that depends on the features, dropping terms that do not depend on the parameters, and check it against the table in Section 2.1.3.3.

4. Compare the parameterizations $\nu(\mathbf{x}) = \mathbf{w}^\top \mathbf{x} + b$ and $\log \nu(\mathbf{x}) = \mathbf{w}^\top \mathbf{x} + b$. Determine for each whether $\nu(\mathbf{x}) > 0$ holds for all inputs and parameters, and state the consequence for the loss of part (c). Write that loss in terms of $(\mathbf{w}, b)$ for the second parameterization.

6. code **The size of the noise ball.** Generate $n = 1{,}000$ examples with two features drawn from $\mathcal{N}(0, 1)$ and labels $y = \mathbf{x}^\top \mathbf{w}^* + \epsilon$, where $\mathbf{w}^* = (2, -3.4)$ and $\epsilon \sim \mathcal{N}(0, 1)$, and compute the least-squares solution $\hat{\mathbf{w}}$ with `lstsq`.

   1. Run minibatch SGD without a bias, starting at $\hat{\mathbf{w}}$, with minibatches of 10 and $\eta = 0.1$ for 20,000 steps. Estimate the mean squared distance $\mathbb{E}\|\mathbf{w} - \hat{\mathbf{w}}\|^2$ of the iterates from $\hat{\mathbf{w}}$.
   2. Repeat for $\eta \in \{0.05, 0.025\}$ and for minibatch sizes 20 and 40. Check that the mean squared distance is roughly proportional to $\eta / |\mathcal{B}|$.
   3. Why does the distance not depend on how far $\hat{\mathbf{w}}$ is from $\mathbf{w}^*$? What would change if the labels had no noise at all?

7. **Composing two linear layers.** Suppose a network with two layers is built by feeding the output of one linear layer, $\mathbf{h} = \mathbf{W}_1 \mathbf{x} + \mathbf{b}_1$, into a second, $\hat{y} = \mathbf{w}_2^\top \mathbf{h} + b_2$. Determine the class of functions of $\mathbf{x}$ that this network can represent and compare it with the class represented by a single linear layer. What property must an operation inserted between the two layers have for the composition to represent a strictly larger class?

8. **Singular design matrix.** ▣ Full column rank makes the least-squares minimizer unique. Suppose instead that $\mathbf{X}^\top \mathbf{X}$ is singular.

   1. Show that a minimizer of $\|\mathbf{y} - \mathbf{X}\mathbf{w}\|^2$ still exists and characterize the set of all minimizers.
   2. Write the singular value decomposition as $\mathbf{X} = \mathbf{U} \Sigma \mathbf{V}^\top$ with orthogonal $\mathbf{U}$ and $\mathbf{V}$ and a rectangular diagonal Σ of nonnegative singular values. The Moore–Penrose pseudoinverse is
   $$\mathbf{X}^\dagger = \mathbf{V} \Sigma^\dagger \mathbf{U}^\top, \tag{2.15}$$
   where $\Sigma^\dagger$ transposes Σ and inverts its nonzero entries. Show that $\mathbf{X}^\dagger \mathbf{y}$ is the minimizer of smallest norm within the set of part (a).
   3. Starting from $\mathbf{w} = \mathbf{0}$, which minimizer does gradient descent converge to? State whether the answer changes for minibatch stochastic gradient descent.

9. **Noise injection and ridge regression.** ▣ Let $\mathbf{X}^\top \mathbf{X}$ be singular or nearly so, and recall that among all minimizers of $\|\mathbf{y} - \mathbf{X}\mathbf{w}\|^2$ the one of smallest norm is $\mathbf{X}^\dagger \mathbf{y}$, where the pseudoinverse $\mathbf{X}^\dagger$ inverts the nonzero singular values of $\mathbf{X}$ and keeps the zero ones at zero.

   1. Perturb the inputs: replace $\mathbf{X}$ by $\mathbf{X} + \mathbf{Z}$, where the entries of $\mathbf{Z}$ are independent draws from $\mathcal{N}(0, \gamma^2)$. Compute $\mathbb{E}[(\mathbf{X} + \mathbf{Z})^\top (\mathbf{X} + \mathbf{Z})]$ and determine whether it is invertible. Express the expected loss $\mathbb{E}\|\mathbf{y} - (\mathbf{X} + \mathbf{Z})\mathbf{w}\|^2$ in terms of $\mathbf{X}$, $\mathbf{y}$, $\mathbf{w}$ and γ.
   2. The standard remedy for a (near-)singular $\mathbf{X}^\top \mathbf{X}$ is to add $\lambda' \mathbf{I}$ before inverting. Show that
   $$\mathbf{w}_{\lambda'} = (\mathbf{X}^\top \mathbf{X} + \lambda' \mathbf{I})^{-1} \mathbf{X}^\top \mathbf{y} \tag{2.16}$$
   is well defined for every $\lambda' > 0$ and that it minimizes $\|\mathbf{y} - \mathbf{X}\mathbf{w}\|^2 + \lambda' \|\mathbf{w}\|^2$, the ℓ_2 penalty of Section 2.7 written for the summed loss. Find the λ' for which this objective equals the expected loss of part (a). Compare the limit of $\mathbf{w}_{\lambda'}$ as $\lambda' \to 0$ with $\mathbf{X}^\dagger \mathbf{y}$.

2.2 Synthetic Data

We have derived linear regression, but we have not trained it yet. That takes data, a model, and a training loop in code, and we begin with the data. Which data should the first run use? Real datasets are what we ultimately care about, but they make a poor first test. Suppose that we fit a linear model to house prices like those of Section 2.1 and its predictions are poor. Three explanations compete: the model may be misspecified, the optimization algorithm or its implementation may be flawed, or the data may be pathological. On real data we cannot easily tell which of the three it is.

Synthetic data removes the ambiguity by construction. If we generate the data ourselves, we know the true weights $\mathbf{w}^*$, the true bias b^*, and the distribution of the noise. A model of the right form then has a known target, and any systematic failure to recover $\mathbf{w}^*$ and b^*, beyond what the noise explains, points to the algorithm or its implementation. It is good practice, and not only in this chapter, to try a new learning method first on a problem whose answer we chose ourselves, and only then on real data.

This section builds such a test for linear regression. We generate examples from a linear function with the true parameters $\mathbf{w}^* = (2, -3.4)$ and $b^* = 4.2$, and add a little Gaussian noise. We then work out how close a correct fit can come to these parameters. This is the step that matters most, since it turns the comparison into a test with a threshold. The next section trains a linear model on these data and holds its estimates to the threshold.

2.2.1 Generating the Data

We draw the features of each example independently from a standard normal distribution, with mean 0 and variance 1, and stack the feature vectors as the rows of the design matrix $\mathbf{X}$. Two features keep the true parameters easy to compare; nothing below depends on that choice. Each label is the value of the true linear function plus noise:

$$\mathbf{y} = \mathbf{X}\mathbf{w}^* + b^* + \epsilon. \tag{2.17}$$

The bias b^* is added to every entry, and the entries of the noise vector ϵ are drawn independently from a normal distribution with mean 0 and standard deviation $\sigma = 0.01$. Each label thus lies close to the plane $\mathbf{w}^{*\top}\mathbf{x} + b^*$, typically within a few hundredths. This is the model of Section 2.1: a linear function with Gaussian noise, for which minimizing the squared loss is the same as maximizing the likelihood (Section 2.1.3). A linear model trained with the squared loss is therefore correctly specified for these data, and the first of our three explanations is ruled out in advance.

We write the generator as a function, `synthetic_data`, and save it in the `d2l` library, since later sections use it again. It takes the true parameters `w` and `b`, the noise level `noise`, and the number `n` of examples, and returns the design matrix and the labels. The function draws the features first and the noise second.

```
def synthetic_data(w, b, noise, n):
    """n examples with standard normal features and labels Xw + b + noise."""
    X = d2l.randn(n, len(w))
    eps = d2l.randn(n, 1) * noise
    return X, d2l.matmul(X, d2l.reshape(w, (-1, 1))) + b + eps
```

We draw 2,000 examples with the parameters chosen above: the first 1,000 for training and the other 1,000 for validation. The validation examples come from the same process but are not used to fit the model. The loss on them shows how well the model predicts examples it has not seen, a question we take up in Section 2.6. The next section draws its data in the same way.

```
w_star, b_star, noise = d2l.tensor([2, -3.4]), 4.2, 0.01
X, y = synthetic_data(w_star, b_star, noise, 2000)
```

Each row of `X` holds the two features of one example, and each row of `y` its label. It is instructive to look at one of them. For the first example we also compute the noise, the label minus the noise-free value $\mathbf{w}^{*\top}\mathbf{x} + b^*$.

```
x, label = X[0], y[0, 0]
eps = label - (d2l.reduce_sum(w_star * x) + b_star)
print(f'features: ({float(x[0]):.4f}, {float(x[1]):.4f})')
print(f'label: {float(label):.4f}, noise: {float(eps):.4f}')
```

```
features: (0.8964, 0.3222)
label: 4.8900, noise: -0.0073
```

The label misses the value of the linear function only by the noise, and by no more than the few hundredths that we expected.

2.2.2 Recovering the Parameters

A test needs a threshold. Even a perfect implementation does not recover $\mathbf{w}^*$ and b^* exactly, because the noise moves the labels and, with them, the minimizer of the loss. How far does it move? The answer follows from the normal equations of Section 2.1.2.1. As there, we append a column of ones to the n training examples and the bias to the weights: $\tilde{\mathbf{X}} = [\mathbf{X}, \mathbf{1}]$ has three columns, the true parameters are $\tilde{\mathbf{w}}^* = (2, -3.4, 4.2)$, and $\mathbf{y} = \tilde{\mathbf{X}}\tilde{\mathbf{w}}^* + \epsilon$. The least-squares solution $\hat{\tilde{\mathbf{w}}}$ solves the normal equations $\tilde{\mathbf{X}}^\top\tilde{\mathbf{X}}\hat{\tilde{\mathbf{w}}} = \tilde{\mathbf{X}}^\top\mathbf{y}$. Substituting the labels gives $\tilde{\mathbf{X}}^\top\tilde{\mathbf{X}}(\hat{\tilde{\mathbf{w}}} - \tilde{\mathbf{w}}^*) = \tilde{\mathbf{X}}^\top\epsilon$, and therefore

$$\hat{\tilde{\mathbf{w}}} - \tilde{\mathbf{w}}^* = (\tilde{\mathbf{X}}^\top\tilde{\mathbf{X}})^{-1}\tilde{\mathbf{X}}^\top\epsilon. \tag{2.18}$$

The *estimation error* $\hat{\tilde{\mathbf{w}}} - \tilde{\mathbf{w}}^*$ is a linear function of the noise, so without noise least squares would recover the parameters exactly.

In general the inverse of $\tilde{\mathbf{X}}^\top\tilde{\mathbf{X}}$ is awkward to reason about. For our data, luckily, the matrix is simple. Each of its entries is a sum over the n examples of a product of two columns, so $\tilde{\mathbf{X}}^\top\tilde{\mathbf{X}}/n$ is a matrix of averages. On the diagonal we average the square of the constant 1 or of a standard normal feature, and either has mean 1. Off the diagonal we average the product of a feature with the constant or with the other, independent feature, which has mean 0. An average over $n = 1{,}000$ examples lies close to its mean, here to within a few percent. With $\tilde{\mathbf{X}}^\top\tilde{\mathbf{X}} \approx n\mathbf{I}$, the estimation error becomes an average over the training examples,

$$\hat{\tilde{\mathbf{w}}} - \tilde{\mathbf{w}}^* \approx \frac{1}{n}\tilde{\mathbf{X}}^\top \epsilon = \frac{1}{n}\sum_{i=1}^{n} \epsilon^{(i)} \tilde{\mathbf{x}}^{(i)}, \tag{2.19}$$

where $\tilde{\mathbf{x}}^{(i)} = (\mathbf{x}^{(i)}, 1)$ is the i-th row of $\tilde{\mathbf{X}}$. Each coordinate of the estimation error averages n independent terms $\epsilon^{(i)}\tilde{x}_j^{(i)}$, each with mean 0 and variance σ^2, since the noise is independent of the features and $\tilde{x}_j^2$ has mean 1. Such an average has standard deviation $\sigma/\sqrt{n}$, which Section 1.6 calls its standard error. With $\sigma = 0.01$ and $n = 1{,}000$ training examples, every coordinate of $\hat{\tilde{\mathbf{w}}}$, the bias included, misses its true value by about $0.01/\sqrt{1000} \approx 3 \times 10^{-4}$.

We check this prediction by solving the least-squares problem on the training examples with `lstsq`, as in Section 2.1.2.1. In the code, `X1` is $\tilde{\mathbf{X}}$ and `w_hat` is $\hat{\tilde{\mathbf{w}}}$.

```
n = 1000                                        # The training examples
X1 = torch.cat((X[:n], torch.ones(n, 1)), 1)    # A column of ones for b
w_hat = torch.linalg.lstsq(X1, y[:n]).solution[:, 0]
```

We print the estimation errors of the two weights and of the bias, and the standard error that we predicted for each.

```
e_w = d2l.numpy(w_hat[:2] - w_star)
print(f'error in w: ({e_w[0]:.1e}, {e_w[1]:.1e})')
print(f'error in b: {float(w_hat[2] - b_star):.1e}')
print(f'standard error: {noise / n ** 0.5:.1e}')
```

```
error in w: (4.1e-04, 4.7e-05)
error in b: 2.1e-04
standard error: 3.2e-04
```

All three estimation errors are at most a few times 10^{-4}, the size of the standard error, as predicted. An error above 10^{-3}, more than three standard errors, would be rare. This gives the next section its threshold. Of course, minibatch stochastic gradient descent does not solve the normal equations. It approaches the same least-squares estimate, and its random minibatches keep it fluctuating around that estimate by a similar amount (the noise ball of Section 2.1). A correct implementation, trained until the loss has converged, should therefore land within about 10^{-3} of the true parameters. An error of 0.1 cannot be blamed on the noise: it points to a bug, or to an optimization that has not converged. The threshold depends on the data: it shrinks with more examples and less noise (exercise 2), and it assumes features of unit scale (exercise 3).

Note, however, how little the test claims. Passing it tells us that an implementation recovers the parameters it should recover, but not how the method behaves when the model does not match the data, which is the usual case with real data (exercise 4).

2.2.3 Summary

A poor result on real data cannot tell a bug from a wrong model; a synthetic dataset can. For labels generated from a linear function with known parameters plus Gaussian noise, a

linear model with the squared loss is correctly specified. The noise moves even the exact least-squares estimate away from the truth, by about the standard error $\sigma/\sqrt{n}$, here 3×10^{-4}, so a correct implementation trained to convergence should land within about 10^{-3} of the true parameters. The test says nothing about data that a linear model does not fit.

2.2.4 Exercises

1. code **Reproducibility.** Two calls of `synthetic_data` with the same arguments return different `X` and `y`. Make them identical by fixing the seed with `torch.manual_seed`. State where the call must be placed relative to the draws of `X` and `eps`, and explain why seeding once at the top of a notebook does not make the second dataset equal to the first.

2. code **Recovery under noise.** ⊠ Repeat the least-squares fit of Section 2.2.2 on 1,000 independent datasets for each setting below, and measure the standard deviation of the estimation error $\hat{w}_1 - w_1^*$ of the first weight.

   1. Vary `noise` over $\{0.001, 0.01, 0.1, 1\}$ with 1,000 training examples, and the number n of training examples over $\{10, 100, 1{,}000, 10{,}000\}$ with `noise=0.01`. Plot both measurements against the prediction $\sigma/\sqrt{n}$ on logarithmic axes.
   2. Find the smallest n at which the measurement agrees with $\sigma/\sqrt{n}$ to within 10%, and say which step of the derivation fails for fewer examples.
   3. Derive $\mathrm{Cov}(\hat{\tilde{\mathbf{w}}}) = \sigma^2(\tilde{\mathbf{X}}^\top\tilde{\mathbf{X}})^{-1}$ from the expression for $\hat{\tilde{\mathbf{w}}} - \tilde{\mathbf{w}}^*$ in this section. Then generate correlated features $x_2 = \rho x_1 + \sqrt{1-\rho^2}\, z$, with x_1 and z independent and standard normal, for $\rho \in \{0, 0.9, 0.99\}$. Predict the standard error of $\hat{w}_1$ from this formula, check it numerically, and explain why the predictions $\hat{\tilde{\mathbf{w}}}^\top \tilde{\mathbf{x}}$ on new examples drawn from the same distribution remain accurate when the weights do not.

3. code **A feature on another scale.** Draw the first feature with standard deviation 10 instead of 1, and set $w_1^* = 0.2$, so that the labels are distributed as before.

   1. Predict $\tilde{\mathbf{X}}^\top\tilde{\mathbf{X}}/n$, say which step of the derivation in this section no longer holds, and predict the standard errors of $\hat{w}_1$ and $\hat{w}_2$.
   2. Check both predictions by repeating the least-squares fit on many independent datasets.
   3. A bug moves $\hat{w}_1$ by 5×10^{-4}. Does the threshold of 10^{-3} catch it? Propose a threshold for each weight that does not depend on the scale of its feature.

4. code **A model that does not fit.** ⊠ Add the term $c\, x_1^2$ with $c = 0.5$ to every label, so that the labels no longer follow a linear function of the features, and fit the linear model by least squares as in this section.

   1. Using $\mathbb{E}[x_1^2] = 1$ and $\mathbb{E}[x_1^3] = 0$, predict which of $\hat{w}_1$, $\hat{w}_2$, and $\hat{b}$ end up away from the true values, and by about how much.
   2. Check the prediction for $n \in \{100, 1{,}000, 10{,}000\}$ training examples. Which estimation errors shrink as n grows, which do not, and how do the shrinking ones compare with $\sigma/\sqrt{n}$?
   3. Would the threshold of 10^{-3} declare a correct implementation faulty on these data? Explain what passing or failing the test of this section can tell you when the model does not fit the data.

5. **The loss floor and its fluctuation.** The next section judges a training run by its validation loss, the average of $\frac{1}{2}(\hat{y} - y)^2$ over the 1,000 validation examples.
    1. Show that a model with the true parameters has expected loss $\sigma^2/2$ on a new example, and that no other parameters have a smaller one. This is the floor that a correct training run levels off at.
    2. For $\epsilon \sim \mathcal{N}(0, \sigma^2)$ one has $\mathbb{E}[\epsilon^4] = 3\sigma^4$. Compute the standard deviation of the loss $\frac{1}{2}\epsilon^2$ of one example, and from it the standard error of the validation loss of the true parameters over $n' = 1{,}000$ examples, as a fraction of the floor.
    3. code Check both numbers by evaluating the true parameters on many independent validation sets. Then use the result of part (a) to find the smallest parameter error that raises the expected loss by one standard error of the validation loss, and compare it with the threshold of 10^{-3}.

2.3 Implementation from Scratch

So far linear regression exists on paper, next to a dataset whose true parameters we know (Section 2.2) and a threshold of about 10^{-3} that a correct fit must meet. We now turn the derivation into a program. The trouble with training programs is that they can be wrong without failing. Subtracting a label vector of shape $(32,)$ from predictions of shape $(32, 1)$, for example, raises no error, because broadcasting forms the 32×32 matrix of all pairwise differences. Training on the loss computed from it lowers that loss steadily while learning the wrong weights. Nothing on the screen looks amiss. Such bugs show only when we compare the result with what a correct program must produce.

We therefore write the four components of training ourselves, using nothing but tensors and automatic differentiation: the linear model, the squared loss, minibatch stochastic gradient descent, and the training loop that runs them. Writing each component explicitly shows what it computes and what it passes to the next one. Every part of the program is a plain function, and the training loop is a dozen lines. Since we train on data whose origin we know, we can predict what a correct implementation must achieve and then check it: the loss must fall to a floor set by the noise, and the learned parameters must come within the threshold of the true ones.

2.3.1 Model, Loss, and Gradient

We draw the data as in Section 2.2, with `d2l.synthetic_data`: 2,000 examples, of which the first 1,000 are for training and the other 1,000 for validation.

```
w_star, b_star, noise = d2l.tensor([2, -3.4]), 4.2, 0.01
X, y = d2l.synthetic_data(w_star, b_star, noise, 2000)
X_train, y_train, X_val, y_val = X[:1000], y[:1000], X[1000:], y[1000:]
```

2.3.1.1 The Parameters

The model has two parameters, the weight vector $\mathbf{w} \in \mathbb{R}^2$ and the bias b. We initialize the weights with random numbers drawn from a normal distribution with mean 0 and standard deviation 0.01, and the bias with 0. For linear regression the starting point matters little: the

loss is a quadratic function of the parameters with a single minimum, which gradient descent reaches from any start (exercise 1). For deep networks the scale of the initial weights matters a great deal, which is why the standard deviation is an argument, `sigma`. (It has nothing to do with the noise level σ of the data. That both are 0.01 is a coincidence.) We will start from fresh parameters several times, so `init_params` creates them.

`requires_grad=True` asks PyTorch to record the operations on a tensor, as in Section 1.5, so that `backward` can store the gradient in its `grad` attribute.

```
def init_params(sigma=0.01):
    w = torch.normal(0, sigma, (2, 1), requires_grad=True)
    b = torch.zeros(1, requires_grad=True)
    return w, b
```

2.3.1.2 The Forward Pass

The forward pass computes the predictions for a minibatch, $\hat{\mathbf{y}} = \mathbf{X}\mathbf{w} + b$. For a minibatch of 32 examples, $\mathbf{X}$ is the 32×2 matrix whose rows are the examples' features. Its product with the weights, stored as a 2×1 matrix, is a 32×1 column of weighted sums, and broadcasting adds the scalar b to each of its entries (Section 1.1.3.2).

```
def linreg(X, w, b):
    """The linear regression model."""
    return d2l.matmul(X, w) + b
```

2.3.1.3 The Loss

The loss of one example is half its squared error, $\frac{1}{2}(\hat{y} - y)^2$, as in Section 2.1, and the loss of a minibatch is the mean over its examples. Before subtracting, the loss reshapes the labels into a column of shape $(|\mathcal{B}|, 1)$, the shape of the predictions. Our labels already have that shape, so the reshape is a precaution. It costs nothing, and it rules out the bug of the opening. What that bug does to training is worked out in exercise 2.

```
def squared_loss(y_hat, y):
    """Half the squared error, averaged over the minibatch."""
    return d2l.reduce_mean((y_hat - d2l.reshape(y, (-1, 1))) ** 2 / 2)
```

2.3.1.4 The Gradient

Before we hand the loss to automatic differentiation, we write down the gradient that it must return. Otherwise we would have nothing to check the result against. Section 2.1.2.3 derived it with the chain rule: for one example with error $r = \hat{y} - y$,

$$\frac{\partial l}{\partial \mathbf{w}} = (\hat{y} - y)\,\mathbf{x} \qquad \text{and} \qquad \frac{\partial l}{\partial b} = \hat{y} - y. \tag{2.20}$$

In words, *the gradient is the error-weighted input*: each weight receives the error times its feature, and the bias the error itself. For a minibatch $\mathcal{B}$, as in the forward pass, $\mathbf{X}$ and $\mathbf{y}$ hold the features and labels of its examples, and we collect their errors in the vector $\mathbf{r} =$

$\mathbf{X}\mathbf{w} + b - \mathbf{y}$. The *minibatch loss* $L_{\mathcal{B}}$ is the average of the per-example losses over $\mathcal{B}$, so its gradient is the average of the per-example gradients, and these averages become matrix products,

$$\nabla_{\mathbf{w}} L_{\mathcal{B}} = \frac{1}{|\mathcal{B}|}\mathbf{X}^\top \mathbf{r} \qquad \text{and} \qquad \frac{\partial L_{\mathcal{B}}}{\partial b} = \frac{1}{|\mathcal{B}|}\sum_{i \in \mathcal{B}} r^{(i)}.$$

One matrix–vector product computes the weight gradient of the whole minibatch. It is the gradient $2\mathbf{X}^\top \mathbf{r}$ of the line model in Section 1.4, divided by $2|\mathcal{B}|$, because our loss is half the mean of the squared errors instead of their sum.

Does automatic differentiation return these gradients? To find out, we take a minibatch of 32 training examples and fresh parameters.

```
Xb, yb = X_train[:32], y_train[:32]
w, b = init_params()
```

We compute the gradient with respect to $(\mathbf{w}, b)$ once by automatic differentiation and once with the formulas above, and print the gradient and the largest difference between the two.

```
squared_loss(linreg(Xb, w, b), yb).backward()
with torch.no_grad():
    r = linreg(Xb, w, b) - yb                # One error per example
    by_hand = torch.cat([Xb.T @ r, r.sum(0, keepdim=True)]) / len(yb)
    by_autograd = torch.cat([w.grad, b.grad[None]])
print('gradient:', d2l.numpy(by_autograd).reshape(-1).round(4))
print(f'largest difference: {float(abs(by_hand - by_autograd).max()):.1g}')
```

```
gradient: [-3.5987  4.338  -5.0454]
largest difference: 0
```

The largest difference is zero or at most about 10^{-6}: the two computations agree up to the rounding error of float32 arithmetic. This is the gradient that the backward pass computes in every training step.

2.3.2 Training

2.3.2.1 The Update

Strictly speaking, we need no optimizer at all: linear regression can be solved in closed form, with the normal equations of Section 2.1. We train it with minibatch stochastic gradient descent nevertheless, because that is how every later model in the book is trained, and because here, for once, its result can be checked against known parameters. Each step draws a minibatch $\mathcal{B}$ and moves the parameters against the gradient of its loss,

$$(\mathbf{w}, b) \leftarrow (\mathbf{w}, b) - \eta \nabla_{(\mathbf{w}, b)} L_{\mathcal{B}}, \tag{2.21}$$

where the learning rate $\eta > 0$ sets the size of the step. Since the loss is a mean over the minibatch, η needs no change when the minibatch size changes (Section 2.1.2.3).

One step of minibatch SGD consists of four operations, which the training loop below performs in this order:

1. the *forward pass* computes the predictions and the loss of the minibatch, while the framework records the operations;
2. the *backward pass* computes the gradient of the loss from that record;
3. the *update* moves the parameters against the gradient, without being recorded;
4. any gradient that the framework keeps is cleared.

These four operations are the whole algorithm. Two rules fix their order. The gradient used in an update must belong to the current minibatch alone, so a gradient that the framework keeps must be cleared before the next backward pass. And the update must not be recorded: it is not part of the loss, and recording it would make the next backward pass differentiate through every earlier update.

2.3.2.2 Minibatches

The minibatches must be drawn at random, or their average gradient is no longer an unbiased estimate of the full gradient (Section 2.1.2.3); minibatches taken in a fixed order, say sorted by label, would bias every step. The usual way is to shuffle the training examples at the start of every epoch, one pass through the training data, and to cut the shuffled sequence into consecutive pieces of `batch_size` examples. Every example is then used once per epoch, and the minibatches differ from one epoch to the next.

The function `minibatches` does this. Because its body contains `yield`, calling it returns a generator, an object that runs the body piece by piece as a loop iterates over it: each `yield` hands the loop one minibatch, a pair of features and labels.

```
def minibatches(X, y, batch_size):
    """Minibatches of (X, y), in a new random order on every call."""
    indices = np.random.permutation(len(X))
    for i in range(0, len(indices), batch_size):
        batch = indices[i : i + batch_size]
        yield X[batch], y[batch]
```

With 1,000 training examples and minibatches of 32, an epoch consists of $\lceil 1000/32 \rceil = 32$ minibatches: 31 full ones and a last one with the remaining 8 examples. We count the minibatches of one pass over a generator, and then try a second pass over the same generator.

```
loader = minibatches(X_train, y_train, 32)
sizes = [len(y) for X, y in loader]
print(f'{len(sizes)} minibatches, the last of size {sizes[-1]}')
print(f'second pass: {len([y for X, y in loader])} minibatches')
```

```
32 minibatches, the last of size 8
second pass: 0 minibatches
```

The count matches. The second pass, however, yields nothing: a generator can be iterated only once, and after one pass it is exhausted, without an error. The training loop below therefore calls `minibatches` anew in every epoch.

2.3.2.3 The Training Loop

The training loop repeats the step on every minibatch of an epoch, epoch after epoch. After each epoch it measures the loss on the 1,000 validation examples, on which the model never trains. `train` starts from fresh parameters, trains with learning rate `lr`, and returns the trained parameters and the validation loss after every epoch. The comments mark the operations of a step; the other lines run the two loops and record the validation loss.

PyTorch adds the gradient of each backward pass to what `.grad` already holds (Section 1.5), so `train` zeroes `.grad` after each update. The update runs inside `torch.no_grad()`, since PyTorch refuses to change a tensor that requires gradients in place while it records.

```
def train(lr, num_epochs=10, batch_size=32):
    w, b = init_params()
    val_losses = []
    for epoch in range(num_epochs):
        for Xb, yb in minibatches(X_train, y_train, batch_size):
            loss = squared_loss(linreg(Xb, w, b), yb)  # forward pass, loss
            loss.backward()                            # backward pass
            with torch.no_grad():                      # not recorded:
                for param in (w, b):
                    param -= lr * param.grad           # update
                    param.grad.zero_()                 # clear
        with torch.no_grad():
            val_losses.append(squared_loss(linreg(X_val, w, b), y_val).item())
    return w, b, val_losses
```

2.3.3 Checking the Implementation

2.3.3.1 The Expected Loss

It is tempting to run the training loop now and see what comes out. But then we could not tell a correct loss curve from a merely plausible one. So before training, we predict what a correct implementation must produce. The label of a new example is $y = \mathbf{x}^\top \mathbf{w}^* + b^* + \epsilon$, so the error of parameters $(\mathbf{w}, b)$ on it is $r = \mathbf{x}^\top(\mathbf{w} - \mathbf{w}^*) + (b - b^*) - \epsilon$. The two features of the synthetic data are independent standard normal variables, and the noise ϵ is independent of them. All three have mean zero, each feature has variance 1, and the product of any two of them averages to zero. Squaring r and taking the expectation therefore leaves only the squared terms,

$$\mathbb{E}\left[\tfrac{1}{2}r^2\right] = \tfrac{1}{2}\|\mathbf{w} - \mathbf{w}^*\|^2 + \tfrac{1}{2}(b - b^*)^2 + \tfrac{1}{2}\sigma^2. \tag{2.22}$$

The expected loss is half the squared distance between the parameters and the true ones, plus a floor of $\sigma^2/2 = 5 \times 10^{-5}$. No choice of parameters gets below the floor, since even the true parameters leave the noise ϵ unexplained. The validation loss, an average over 1,000 examples that the model does not train on, estimates the expected loss.

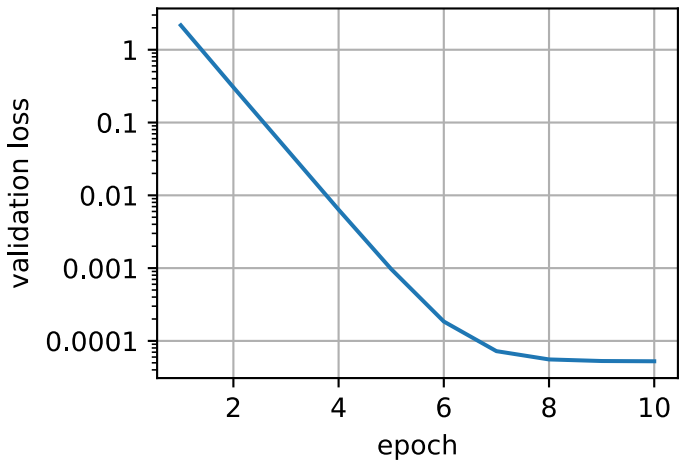
The equation also predicts how fast training approaches the floor. We call $(\mathbf{w} - \mathbf{w}^*, b - b^*)$ the *parameter error*; for the least-squares solution it is the estimation error of Section 2.2.2. The gradient of the expected loss with respect to $(\mathbf{w}, b)$ is the parameter error itself, so a step along this gradient with learning rate η multiplies the parameter error by $1 - \eta$, and the *excess loss* above the floor, half its square, by $(1 - \eta)^2$. The minibatch gradient equals

this gradient only on average, so the rule gives an estimate, not an exact value. At $\eta = 0.03$ an epoch of 32 steps should therefore lower the excess loss by a factor of about 7. Training starts with weights close to zero and a zero bias, so the excess loss starts at half the squared length of $(\mathbf{w}^*, b^*)$, about 16.6, and the rule predicts it for every epoch from there. We compare the prediction with the measured loss, epoch by epoch, once the model is trained.

2.3.3.2 The Training Run

We train for ten epochs with learning rate 0.03. The loss falls by more than five orders of magnitude, so we plot the validation loss after each epoch on a logarithmic axis, on which a constant factor per epoch is a straight line.

```
w, b, val_losses = train(lr=0.03)
d2l.plot(list(range(1, 11)), val_losses, xlabel='epoch',
         ylabel='validation loss', yscale='log')
```



On the plot the loss falls along a straight line, by a factor of about 7 per epoch, and levels off near 5×10^{-5} after seven or eight epochs. To compare the rate in numbers, we print for the first seven epochs the validation loss above the floor next to the predicted excess loss.

```
floor = noise ** 2 / 2                              # sigma^2 / 2
excess = (float(d2l.reduce_sum(w_star ** 2)) + b_star ** 2) / 2  # 16.6
for epoch, loss in enumerate(val_losses[:7], start=1):
    excess *= (1 - 0.03) ** (2 * 32)                # 32 steps per epoch
    print(f'epoch {epoch}: measured {loss - floor:.1e}, '
          f'predicted {excess:.1e}')
```

```
epoch 1: measured 2.2e+00, predicted 2.4e+00
epoch 2: measured 3.1e-01, predicted 3.4e-01
epoch 3: measured 4.4e-02, predicted 4.8e-02
epoch 4: measured 6.3e-03, predicted 6.8e-03
epoch 5: measured 9.2e-04, predicted 9.7e-04
epoch 6: measured 1.3e-04, predicted 1.4e-04
epoch 7: measured 2.2e-05, predicted 2.0e-05
```

Measured and predicted excess agree closely in the first epochs and drift apart slowly, by a factor of up to about 3 or 4 at the seventh. The prediction takes the matrix $\mathbf{H}$ of exercise 4, the average of $(\mathbf{x}, 1)(\mathbf{x}, 1)^\top$ over the training set, to be exactly the identity. For 1,000 examples its eigenvalues lie within about 10% of 1, and after a few epochs the smallest of them sets the rate, so the run falls behind the prediction by a little more with every epoch. The prediction holds, in rate and in floor.

2.3.3.3 Recovered Parameters

A loss at the floor is good news, but it is indirect evidence. Because we generated the data ourselves, we can look at the parameters themselves and compare them with the true ones. We print the parameter errors $\hat{\mathbf{w}} - \mathbf{w}^*$ and $\hat{b} - b^*$.

```
w_error = d2l.numpy(d2l.reshape(w, w_star.shape) - w_star)
b_error = d2l.numpy(b - b_star)
print('error in w:', ', '.join(f'{e:.1e}' for e in w_error))
print('error in b:', ', '.join(f'{e:.1e}' for e in b_error))
```

```
error in w: -2.8e-04, 6.2e-04
error in b: -2.4e-04
```

The parameter errors are a few times 10^{-4}, within the threshold of 10^{-3} that Section 2.2.2 set. Could they be much smaller? No. The noise in the 1,000 training labels moves even the exact least-squares fit about 3×10^{-4} away from the true value in each coordinate, the standard error $\sigma/\sqrt{n}$ derived there. Minibatch SGD adds two contributions of similar size. With a constant learning rate its parameters never settle but keep fluctuating in the noise ball of Section 2.1. And ten epochs leave a remainder of the initial parameter error: the 320 steps multiply it by $0.97^{320} \approx 6 \times 10^{-5}$. The bias starts 4.2 below b^*, so about $4.2 \times 0.97^{320} \approx 2.5 \times 10^{-4}$ of that distance remains.

The loss does not reveal errors of this size. By Equation 2.22, a parameter error of 5×10^{-4} in each of the three coordinates raises the expected loss by $\frac{1}{2} \cdot 3 \cdot (5 \times 10^{-4})^2 \approx 4 \times 10^{-7}$, less than 1% of the floor and less than the fluctuation of a validation loss over 1,000 examples. A comparison of parameters detects implementation errors that a loss curve hides, which is why new code is best tested on synthetic data whose parameters we know. Such a comparison is possible only when a single set of parameters minimizes the loss, as here, where no feature is a combination of the others. Deep networks have many parameter settings that make the same predictions, so we judge them by their predictions on new data.

2.3.3.4 The Learning Rate

For steps along the gradient of the expected loss, the factor $1 - \eta$ also tells us what other learning rates do. For $0 < \eta < 1$ every step shrinks the parameter error, the faster the larger η is, and a step with $\eta = 1$ lands on the minimum of the expected loss. For $1 < \eta < 2$ the steps overshoot the minimum, but the parameter error still shrinks, since $|1 - \eta| < 1$. For $\eta > 2$ every step multiplies the parameter error by a number larger than 1 in magnitude, so it grows without bound. Minibatch noise and the finite training set lower this limit a little (exercise 4). We train with four learning rates, a factor of 10 apart, and print the final validation loss of each run.

```
for lr in [0.003, 0.03, 0.3, 3]:
    val_loss = train(lr)[2][-1]
    print(f'lr = {lr:<6} validation loss: {val_loss:.1e}')
```

```
lr = 0.003  validation loss: 2.3e+00
lr = 0.03   validation loss: 5.2e-05
lr = 0.3    validation loss: 5.4e-05
lr = 3      validation loss: nan
```

At $\eta = 0.003$ the parameter error shrinks by only 0.997 per step, and after ten epochs the loss is still about 2 to 3, as the prediction $16.6 \times 0.997^{640} \approx 2.4$ says. At $\eta = 0.3$ the loss reaches the floor within the first epoch. At $\eta = 3$ every step multiplies the parameter error by about -2, and the loss overflows until it ends as `nan`. Note that the limit of 2 is a property of our data, not of gradient descent: it holds because every feature has variance 1. A feature scaled by $s > 1$ multiplies the curvature of the loss along its weight by s^2 and lowers the limit to about $2/s^2$, which is one reason why inputs are standardized (Section 1.2).

All of these checks relied on knowing the true parameters, a luxury that real data never provide; Section 2.6 asks how to judge a model by data it has not seen. First, though, Section 2.4 packages this code into classes, so that later chapters change only the part under study.

2.3.4 Summary

Training linear regression takes four components, each a plain function: parameters with a forward pass, the squared loss, minibatch stochastic gradient descent, and a training loop of a dozen lines. The gradient of the squared loss is the error-weighted input, and automatic differentiation reproduces it up to rounding. A step runs the forward pass and the loss, the backward pass, and an update that is not recorded, and then clears any gradient that the framework keeps. On synthetic data the expected loss is half the squared parameter error plus $\sigma^2/2$, which predicts the floor of the loss, the rate at which training reaches it, and the learning rates that diverge. The learned parameters come within a few times 10^{-4} of the true ones, an error that a loss curve cannot show and a comparison of parameters can.

2.3.5 Exercises

1. code **Initialization at the extremes.** `init_params` draws the weights with standard deviation `sigma=0.01`. Predict whether training reaches the noise floor when the weights are initialized to exactly zero, and when `sigma=1000`. Use Equation 2.22 to estimate how many more epochs the second case needs at learning rate 0.03, then check your estimate.
2. **Shapes in the loss.** Suppose the data loader yielded labels `y` of shape $(|\mathcal{B}|,)$ while the predictions `y_hat` have shape $(|\mathcal{B}|, 1)$, and `squared_loss` computed `(y_hat - y) ** 2 / 2` without reshaping `y`.
    1. Determine whether an error is raised, and find the shape and the entries of `y_hat - y`.
    2. Show that this loss is smallest when the model predicts the mean of the minibatch labels for every example. Determine approximately the weights and the bias that training approaches on the synthetic data, and the value at which the loss levels off.

3. code **Second derivatives.** Compute by hand the Hessian of the minibatch loss with respect to $(\mathbf{w}, b)$, and state which quantities it depends on. Then compute it with automatic differentiation, differentiating through the gradient as in Section 1.5, for one fixed minibatch of the synthetic data, once at initialization and once after training. Compare both results with your formula and with the curvature of Equation 2.22.

4. code **Stability and minibatch size.** ▣ The expected loss predicts that steps along its gradient diverge for $\eta > 2$. Let $\tilde{\mathbf{x}} = (\mathbf{x}, 1)$ be an input with a 1 appended for the bias.

   1. For full-batch gradient descent, show that each step multiplies the difference between the parameters and the least-squares solution by the matrix $\mathbf{I} - \eta\mathbf{H}$, where $\mathbf{H}$ is the average of $\tilde{\mathbf{x}}\tilde{\mathbf{x}}^\top$ over the training set. Compute the eigenvalues of $\mathbf{H}$ and predict the largest stable learning rate.
   2. Train with minibatches of 1,000, 32, and 1 example, giving each run the same number of updates, say 320. For each minibatch size, find the largest learning rate in a grid of your choice at which the final training loss is below 10^{-3}.
   3. Show that a step on a single example multiplies the component of the parameter error along $\tilde{\mathbf{x}}$ by $1 - \eta\|\tilde{\mathbf{x}}\|^2$, and use $\mathbb{E}\|\tilde{\mathbf{x}}\|^2 = 3$ to explain the limit you found for minibatches of one example.

5. **Why reshuffle.** `minibatches` draws a fresh random order in every epoch. Explain which property of the minibatch gradient this preserves. Then construct a fixed ordering of the training set, such as sorting by label, under which minibatch SGD without reshuffling behaves differently, and describe the trajectory of $(\mathbf{w}, b)$ within an epoch and across epochs.

6. code **Robust losses.** ▣ Replace `squared_loss` with the mean absolute error `d2l.reduce_mean(d2l.abs(y_hat - d2l.reshape(y, (-1, 1))))`.

   1. Train on the synthetic data and compare the recovered parameters and the loss curve with those of the squared loss.

   2. Set one training label to $y^{(5)} = 10{,}000$, the corruption used in Section 2.1.1.3, and train with both losses. Record b after every minibatch update, plot it against the iteration for both runs, and explain the pattern of the squared-loss curve, including what determines its final value.

   3. On the corrupted data, replace the loss with the minibatch average of the Huber loss

$$l_\delta(y, y') = \begin{cases} \frac{1}{2}(y - y')^2 & \text{if } |y - y'| \le \delta \\ \delta\left(|y - y'| - \frac{\delta}{2}\right) & \text{otherwise,} \end{cases}$$

      whose quadratic part matches the squared loss of this section. Train for $\delta \in \{1, 10, 10^2, 10^3, 10^4\}$, and plot the distance $|\widehat{b} - b^*|$ of the learned bias from the generating bias $b^* = 4.2$ against δ on logarithmic axes. Explain the shape of the curve from the derivative of l_δ with respect to y', and relate the result to the penalty curves of Figure 2.5.

7. **Uncleared gradients.** ▣ Suppose the gradient buffers accumulate and are never cleared, so that the update at step t subtracts $\eta(\mathbf{g}_1 + \cdots + \mathbf{g}_t)$, the sum of all minibatch gradients so far. For the one-dimensional loss $\frac{h}{2}(w - c)^2$ with $h > 0$, whose minimum lies at $w = c$, and with exact gradients, derive a recurrence for the error

$e_t = w_t - c$, and determine for which values of ηh the error converges to zero. State what this predicts for `train` under the same accumulation.

8. **Planted bug.** A faulty variant of `minibatches` shuffles the rows of `X` and the entries of `y` with two independent permutations. We collect the pairs that it yields in one epoch over the training data, whose parameters are $\mathbf{w}^* = (2, -3.4)$ and $b^* = 4.2$ with noise standard deviation 0.01. We then fit $\hat{\mathbf{w}}$ and $\hat{b}$ to these pairs by least squares.

    1. Predict the approximate values of $\hat{\mathbf{w}}$ and $\hat{b}$, and explain your prediction.
    2. Estimate the mean squared error of this fit on the collected pairs, and compare it with the smallest mean squared error that any predictor can attain on correctly paired data.

2.4 Object-Oriented Design for Implementation

The function `train` of Section 2.3 trained linear regression in a dozen lines. Those lines are written for one model and one dataset: the functions `linreg` and `squared_loss`, the parameters `w` and `b`, and the arrays `X_train` and `X_val` all appear in its body, so any other model would need a copy with a few lines changed. Stripped of these names, the loop reads as follows.

```
for epoch in range(num_epochs):
    for X, y in train_data:                    # data
        loss = loss_fn(net(X), y)              # model and loss
        update(params, gradient(loss, params))  # optimizer
    val_loss = mean_loss(net, val_data)        # model on held-out data
    record(epoch, val_loss)
```

Only the lines with comments depend on the problem; the loop around them never changes. Nearly thirty sections of this book train a model, most of them in several frameworks, so a loop written out in each would exist in close to a hundred copies. A change to how we train, such as clipping very large gradients or lowering the learning rate as training proceeds, would then have to be made in every copy, and the copies would drift apart.

We therefore write the loop once, as a class, and put the parts that change into two more classes, a design borrowed from libraries such as Lightning. A **`Module`** holds a model: its parameters, its predictions (`forward`), its loss, and its *optimizer*, the object that applies an update rule such as minibatch SGD. A **`DataModule`** holds a dataset and serves it in minibatches. A **`Trainer`** runs the loop.

In the pseudocode, the `DataModule` supplies `train_data` and `val_data`, the `Module` supplies `net`, `loss_fn`, and the update rule, and the `Trainer` runs the two `for` loops, computes the gradients, and records the losses (Figure 2.7). Every model in the book is a subclass of `Module`, and every dataset a subclass of `DataModule`.

This section rewrites the code of Section 2.3 in this form and ends by training the same model once more, through the classes. Some arguments and methods exist only for later chapters; we say so where they appear.

```
import collections
```

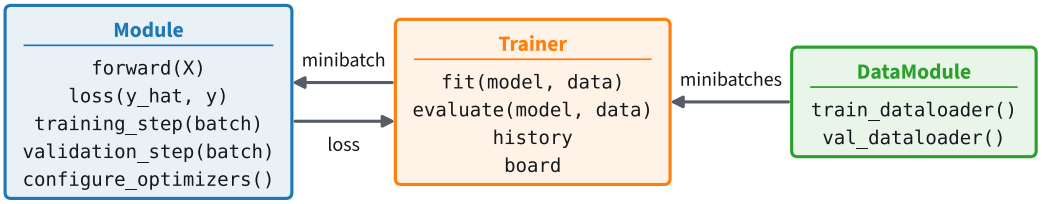


Figure 2.7: The three base classes. The `Trainer` takes minibatches from a `DataModule` and passes each to a `Module`, whose `training_step` returns the loss; the trainer updates the parameters and keeps the epoch means of the losses in its `history`. New models and datasets are written as subclasses of `Module` and `DataModule`.

2.4.1 Utilities

Two helpers make the classes of this book easier to write in a notebook. Neither has anything to do with machine learning. Python defines a class in one block of code, but a class with several methods is easier to explain in several short cells, each next to the prose about it. The decorator `add_to_class` registers a function as a method of a class that already exists, so that one class can be spread over several cells. This is notebook scaffolding, not a design we recommend: in an ordinary Python module, a class is defined in one place.

```
def add_to_class(Class):
    """Register functions as methods in created class."""
    def wrapper(obj):
        setattr(Class, obj.__name__, obj)
        return obj
    return wrapper
```

To see it at work, we create an instance `a` of a class `A` that has no methods other than its constructor.

```
class A:
    def __init__(self):
        self.b = 1

a = A()
```

In a separate cell we register a method `do`. The instance `a`, created before `do` existed, has it too, since Python looks up a method on the class at the moment of the call.

```
@add_to_class(A)
def do(self):
    print('Instance attribute "b" is', self.b)

a.do()
```

```
Instance attribute "b" is 1
```

The second helper removes repetitive code from constructors, which usually store each ar-

gument as an attribute: `self.lr = lr`, `self.batch_size = batch_size`, and so on. Most of these arguments are hyperparameters, such as the learning rate, hence the name of the `d2l` class `HyperParameters`. Its method `save_hyperparameters` stores all arguments of the constructor that calls it as attributes, except those named in `ignore`. It reads them with Python's `inspect` module, which teaches nothing about training, so its body is in the online appendix "Utility Functions and Classes".

```
class B(d2l.HyperParameters):
    def __init__(self, a, b, c):
        self.save_hyperparameters(ignore=['c'])
        print('self.a =', self.a, 'self.b =', self.b)
        print('self.c exists:', hasattr(self, 'c'))

b = B(a=1, b=2, c=3)
```

```
self.a = 1 self.b = 2
self.c exists: False
```

All classes in the rest of this section inherit from `HyperParameters`.

2.4.2 Models

A model in this book is a subclass of `Module`. The subclass creates its parameters in its constructor and defines three methods: `forward` computes the predictions for a minibatch of inputs, `loss` compares predictions with labels, and `configure_optimizers` returns the optimizer. (By default, `forward` applies `self.net`, a network that a subclass may store.) The base class contributes the two steps that the trainer calls, `training_step` and `validation_step`. Besides `HyperParameters`, `Module` inherits from the framework's base class of networks.

That class is `nn.Module`. Calling an instance, `model(X)`, runs `forward`, and the module keeps track of the parameters stored as its attributes (Section 5.1).

```
class Module(nn.Module, d2l.HyperParameters):
    """The base class of models."""
    def loss(self, y_hat, y):
        raise NotImplementedError

    def forward(self, X):
        assert hasattr(self, 'net'), 'Neural network is not defined'
        return self.net(X)

    def training_step(self, batch):
        """The mean loss on one minibatch, or a dict of batch-mean metrics
        with key 'loss'."""
        return self.loss(self(*batch[:-1]), batch[-1])

    def validation_step(self, batch):
        """The mean loss on one minibatch of validation data, or a dict of
        batch-mean metrics with key 'loss'."""
        return self.loss(self(*batch[:-1]), batch[-1])

    def configure_optimizers(self):
```

```
        raise NotImplementedError
```

A minibatch is a tuple whose last element holds the labels and whose other elements hold the inputs. `training_step` calls the model on the inputs and passes the predictions and the labels to `loss`; `validation_step` does the same on validation data. Both steps *return* their result and do nothing else: they do not print, plot, or keep a record. This restriction is the key decision of the design. Nothing in a model refers to the trainer, so any loop, not only the trainer's, can train or evaluate it.

In this chapter a step returns one number, the mean loss over the minibatch. From Section 3.3 on, a step may return a dictionary of such means, with the loss under the key `'loss'` and, say, the accuracy next to it. The function `as_metrics` wraps a single loss in a dictionary, so that the trainer handles both alike.

```
def as_metrics(out):
    """A step returns a loss or a dict of metrics; return a dict."""
    return out if isinstance(out, dict) else {'loss': out}
```

2.4.2.1 Linear Regression as a Module

The model of Section 2.3 becomes a subclass of `Module`. Its constructor creates the parameters as `init_params` did and registers them with the module through the framework's parameter type. Registration lets the framework's tools find the parameters, which are then listed, saved, and moved to a GPU together with the rest of the model. It computes nothing for us: the loss and the update are still the ones we wrote by hand.

`nn.Parameter` wraps a tensor and records the operations on it, as `requires_grad=True` did; assigned as an attribute, it is registered, so `model.parameters()` lists `w` and `b`.

```
class LinearRegressionScratch(d2l.Module):
    """The linear regression model implemented from scratch."""
    def __init__(self, num_inputs, lr, sigma=0.01):
        super().__init__()
        self.save_hyperparameters()
        self.w = nn.Parameter(d2l.normal(0, sigma, (num_inputs, 1)))
        self.b = nn.Parameter(d2l.zeros(1))
```

The forward pass and the loss are `linreg` and `squared_loss` of Section 2.3, now methods that read the parameters from the model.

```
@d2l.add_to_class(LinearRegressionScratch)
def forward(self, X):
    return d2l.matmul(X, self.w) + self.b
```

```
@d2l.add_to_class(LinearRegressionScratch)
def loss(self, y_hat, y):
    l = (y_hat - d2l.reshape(y, (-1, 1))) ** 2 / 2
    return d2l.reduce_mean(l)
```

The update becomes an optimizer class, `SGD`. Its update is the one line of Section 2.3; the rest of the class gives it the interface of the framework's own optimizers, so that Section 2.5 can exchange the two.

As in PyTorch's optimizers, `step` updates the parameters in place and `zero_grad` resets their gradients to zero.

```
class SGD(d2l.HyperParameters):
    """Minibatch stochastic gradient descent."""
    def __init__(self, params, lr):
        self.save_hyperparameters()

    def step(self):
        for param in self.params:
            param -= self.lr * param.grad

    def zero_grad(self):
        for param in self.params:
            if param.grad is not None:
                param.grad.zero_()
```

The model's `configure_optimizers` returns an instance of this class.

```
@d2l.add_to_class(LinearRegressionScratch)
def configure_optimizers(self):
    return SGD([self.w, self.b], self.lr)
```

2.4.3 Data

The base class for data is shorter still. A dataset in this book is a subclass of `DataModule`, whose constructor prepares the data, downloading and preprocessing it if needed. (The argument `root` names the directory for downloaded files, and `num_workers` the number of processes that load data in parallel.) A subclass implements one method, `get_dataloader(train)`. It returns a *data loader*, an object that yields the data one minibatch at a time, as the tuples that the steps of `Module` expect. The other two methods merely call it.

```
class DataModule(d2l.HyperParameters):
    """The base class of data."""
    def __init__(self, root='../data', num_workers=4):
        self.save_hyperparameters()

    def get_dataloader(self, train):
        raise NotImplementedError

    def train_dataloader(self):
        return self.get_dataloader(train=True)

    def val_dataloader(self):
        return self.get_dataloader(train=False)
```

The synthetic data of Section 2.2 become a subclass whose constructor calls `d2l.synthetic_data` for `num_train + num_val` examples, by default 1,000 of each: the first `num_train` for training, the rest for validation.

```
class SyntheticRegressionData(d2l.DataModule):
    """Synthetic data for linear regression."""
    def __init__(self, w, b, noise=0.01, num_train=1000, num_val=1000,
                 batch_size=32):
        super().__init__()
        self.save_hyperparameters()
        self.X, self.y = d2l.synthetic_data(w, b, noise, num_train + num_val)
```

2.4.3.1 A Loader for Every Epoch

What should `get_dataloader` return? The generator `minibatches` of Section 2.3 would not do. The trainer asks a `DataModule` for its loaders once and then iterates over them in every epoch, and a generator serves one pass and then nothing. The trainer would train for one epoch and idle through the rest, without an error.

We need a loader that can be iterated in every epoch, draws a fresh random order on every pass, and reports its number of minibatches through `len`. The loaders that deep learning frameworks provide do more. For data that do not fit in memory, or that must be decoded before use, such as large collections of images, they read examples from disk and prepare the next minibatches in parallel workers while the model trains on the current one. Our dataset fits in memory, so none of this is needed yet.

For data held in memory as tensors, we add the method `get_tensorloader` to `DataModule`. It returns the framework's loader where one suits such data, and otherwise a short class of our own. The validation examples are read in their stored order, since we only average the loss over them.

`TensorDataset` pairs the tensors example by example, and `DataLoader` draws minibatches from it, in a new random order on every pass when `shuffle=True`.

```
@d2l.add_to_class(d2l.DataModule)
def get_tensorloader(self, tensors, train, indices=slice(0, None)):
    tensors = tuple(a[indices] for a in tensors)
    dataset = torch.utils.data.TensorDataset(*tensors)
    return torch.utils.data.DataLoader(dataset, self.batch_size,
                                       shuffle=train)
```

`get_tensorloader` keeps the examples selected by `indices` and shuffles them only if `train` is true. `SyntheticRegressionData` passes it the slice of each split.

```
@d2l.add_to_class(SyntheticRegressionData)
def get_dataloader(self, train):
    i = slice(0, self.num_train) if train else slice(self.num_train, None)
    return self.get_tensorloader((self.X, self.y), train, i)
```

We create the dataset of Section 2.2 and repeat the test of the generator: one pass over the training loader, then a second pass over the same loader.

```
data = SyntheticRegressionData(w=d2l.tensor([2, -3.4]), b=4.2)
```

```
loader = data.train_dataloader()
sizes = [len(y) for X, y in loader]
print(f'{len(sizes)} minibatches, the last of size {sizes[-1]}')
print(f'second pass: {len([y for X, y in loader])} minibatches')
for _ in range(2):  # Each new pass starts in a new random order
    X, y = next(iter(loader))
    print('a new pass starts with labels', d2l.numpy(y[:3, 0]).round(2))
```

```
32 minibatches, the last of size 8
second pass: 32 minibatches
a new pass starts with labels [-0.64 -5.75  3.55]
a new pass starts with labels [-3.35  0.07 -0.03]
```

The first pass yields 32 minibatches, the last of 8 examples, as the generator did. This time, however, the second pass yields all 32 minibatches again, and every pass starts with different examples because the order is drawn anew. The loader also reports its number of minibatches without a pass.

```
len(loader)
```

```
32
```

2.4.4 Training

2.4.4.1 The Training Loop and the History

The `Trainer` class fits the parameters of a `Module` to the data of a `DataModule`. Its constructor takes the number of epochs, `max_epochs`. (The other two arguments can be ignored until they are needed: `num_gpus` in Section 5.7, and `gradient_clip_val`, which limits the size of large gradients, in Section 8.5.) It also creates a progress board, on which the trainer plots the losses; we define it below. Training begins with three preparations: `prepare_data` asks the `DataModule` for its loaders, `prepare_model` stores the model, and `prepare_optimizer` calls the model's `configure_optimizers`. Each is a line or two, and a method of its own so that later sections can override it.

```
class Trainer(d2l.HyperParameters):
    """Fit a Module to a DataModule, keeping a per-epoch history."""
    def __init__(self, max_epochs, num_gpus=0, gradient_clip_val=0):
        self.save_hyperparameters()
        self.board = ProgressBoard(xlabel='epoch', ylabel='loss')

    def prepare_data(self, data):
        self.train_dataloader = data.train_dataloader()
        self.val_dataloader = data.val_dataloader()

    def prepare_model(self, model):
        self.model = model
```

```
    def prepare_optimizer(self, model):
        self.optim = model.configure_optimizers()
```

For later chapters, `prepare_optimizer` also accepts a learning-rate schedule next to the optimizer (Section 9.6).

```
@d2l.add_to_class(Trainer)
def prepare_optimizer(self, model):
    """`configure_optimizers` may also return a learning-rate scheduler."""
    # PyTorch steps a schedule with a separate object (sec_scheduler)
    optim = model.configure_optimizers()
    self.optim, self.scheduler = (optim if isinstance(optim, tuple)
                                  else (optim, None))
```

The method `fit(model, data)` is the outer loop of the pseudocode at the start of this section. After the three preparations it runs `max_epochs` epochs. In each, `fit_epoch` updates the parameters once per minibatch and returns the mean of every quantity that `training_step` reports, and `validate` returns the means over the validation data. The method `record` keeps these means.

```
@d2l.add_to_class(Trainer)
def fit(self, model, data):
    self.prepare_data(data)
    self.prepare_model(model)
    self.prepare_optimizer(model)
    self.history = collections.defaultdict(list)
    self.board.clear()
    self.board.xlim = [0, self.max_epochs]
    for self.epoch in range(self.max_epochs):
        # The training mean averages over the epoch, so it sits mid-epoch
        self.record('train', self.fit_epoch(), x=self.epoch + 0.5)
        if self.val_dataloader is not None:
            self.record('val', self.validate(), x=self.epoch + 1)
        self.board.show(final=False)
    self.board.show()

@d2l.add_to_class(Trainer)
def record(self, split, metrics, x):
    for name in sorted(metrics, key=lambda name: (name != 'loss', name)):
        self.history[f'{split}_{name}'].append(metrics[name])
        self.board.draw(x, metrics[name], f'{split}_{name}')
```

`fit` stores the epoch in `self.epoch`, so that the methods it calls can read it, and skips validation when `val_dataloader` returns `None`.

What is left when training ends? `fit` itself returns nothing. Everything is in `trainer.history`, a dictionary of lists to which `record` appends each mean. Its keys combine the split with the name of the quantity: `'train_loss'`, `'val_loss'`, and, for a classifier, `'val_acc'`. Thus `trainer.history['val_loss'][-1]` is the validation loss after the last epoch. (`record` puts the loss first, so that a curve keeps its color on the board from one model to the next.)

2.4.4.2 Epoch Means

The means that `fit_epoch` and `validate` return are averages over examples, not over minibatches: each minibatch counts in proportion to its size. This matters for the last minibatch of a pass, which is smaller whenever the minibatch size does not divide the number of examples: 1,000 examples in minibatches of 32 make 31 full minibatches and one of 8, and with equal weights per minibatch each of the last 8 examples would count four times as much as the others. The trainer therefore multiplies each value by the number of examples in its minibatch. This requires a step to return a mean and never a sum: a loss summed over a minibatch of 32 examples would enter the average 32 times too large.

`Averager` computes these means. It weights each minibatch by its size, and it keeps the sums on the device, converting them into Python numbers only once per epoch, in `mean`, since every such conversion makes Python wait until a GPU has finished the work queued before it (Section 5.7.2.1). With three parameters on a CPU this saves nothing, and `Averager` may be read as a slightly elaborate way of taking a mean.

```
class Averager:
    """Example-weighted means of batch metrics, summed on the device."""
    def __init__(self):
        self.sums, self.n = {}, 0

    def add(self, metrics, n):
        for name, value in metrics.items():
            self.sums[name] = self.sums.get(name, 0) + value * n
        self.n += n

    def mean(self):
        """One host synchronization per metric; the loss comes first."""
        names = sorted(self.sums, key=lambda name: (name != 'loss', name))
        return {name: float(self.sums[name]) / self.n for name in names}
```

2.4.4.3 Training and Validation Passes

The two passes, `fit_epoch` and `validate`, are the inner loop of `train` in Section 2.3, written against the classes: the minibatches come from the loader, the loss from the model's steps, and the update from the optimizer. Both wrap what a step returns with `as_metrics` and average it with `Averager`. Each minibatch first passes through `prepare_batch`, a hook that a later chapter overrides to move minibatches to a GPU; here it returns the minibatch unchanged.

```
@d2l.add_to_class(d2l.Trainer)
def prepare_batch(self, batch):
    return batch
```

The loop that trains every model in the book is longer than linear regression needs. Its algorithm is the step of Section 2.3: forward pass and loss, backward pass, update, and a clearing of gradients where the framework keeps them. The other lines prepare the minibatch, average its metrics, or are hooks for later chapters, and none of them changes the result of training here.

The two `if` statements are the hooks, for gradient clipping and a learning-rate schedule. `detach()` keeps the running sum of losses from holding on to the computation graph of every

minibatch, and `train()` and `eval()` matter only for layers we have not met yet.

```
@d2l.add_to_class(d2l.Trainer)
def fit_epoch(self):
    self.model.train()
    metrics = d2l.Averager()
    for batch in map(self.prepare_batch, self.train_dataloader):
        out = d2l.as_metrics(self.model.training_step(batch))
        self.optim.zero_grad()
        out['loss'].backward()
        if self.gradient_clip_val > 0:  # To be discussed later
            self.clip_gradients(self.gradient_clip_val, self.model)
        with torch.no_grad():
            self.optim.step()
        if self.scheduler is not None:
            self.scheduler.step()
        metrics.add({k: v.detach() for k, v in out.items()}, len(batch[-1]))
    return metrics.mean()

@d2l.add_to_class(d2l.Trainer)
def validate(self):
    self.model.eval()
    metrics = d2l.Averager()
    with torch.no_grad():
        for batch in map(self.prepare_batch, self.val_dataloader):
            metrics.add(d2l.as_metrics(self.model.validation_step(batch)),
                        len(batch[-1]))
    return metrics.mean()
```

2.4.4.4 The Progress Board

`record` also adds each mean to the progress board, which `fit` redraws after every epoch so that we can watch training in a notebook. The class `ProgressBoard` keeps one list of points per curve: `draw(x, y, label)` appends the point `(x, y)` to the curve named `label`, `show` redraws all curves, and `clear` forgets them. The drawing code has nothing to teach us here and is in the online appendix "Utility Functions and Classes". The board can be configured before training, for instance with `trainer.board.yscale = 'log'` for a logarithmic loss axis, or with `trainer.board.display = False` to train without plotting.

Before we read our first pair of training curves, note that they are computed with different parameters. The validation mean uses the parameters at the end of the epoch, while the training mean averages over minibatches seen while the parameters changed. `fit` therefore draws the training mean of each epoch at its middle, at 0.5, 1.5, and so on, and the validation mean at its end, at 1, 2, and so on.

```
class ProgressBoard(d2l.HyperParameters):
    """Curves that grow during training; `show` redraws them."""
    def __init__(self, xlabel=None, ylabel=None, xlim=None, ylim=None,
                 xscale='linear', yscale='linear', figsize=(3.5, 2.5),
                 display=True):
        self.save_hyperparameters()
```

```
        self.clear()

    def clear(self):
        """Forget all curves; the next `show` starts a new figure."""
        self.data, self.handle = {}, None

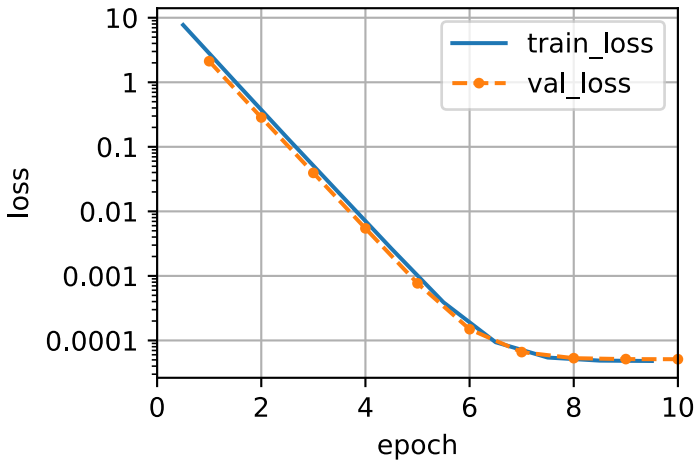
    def draw(self, x, y, label):
        """Append the point (x, y) to the curve called `label`."""
        xs, ys = self.data.setdefault(label, ([], []))
        xs.append(x)
        ys.append(y)

    def show(self, final=True):
        raise NotImplementedError
```

2.4.5 Training Linear Regression Again

We now train the model of Section 2.3 once more, through the classes, on the dataset `data` created above. The arguments of `train` have moved to the classes that own them: the learning rate 0.03 to the model, the minibatch size 32 to the data (it is the default), and the ten epochs to the trainer. We use `d2l.Trainer`, the class of this section together with the drawing code of the online appendix "Utility Functions and Classes".

```
model = LinearRegressionScratch(2, lr=0.03)
trainer = d2l.Trainer(max_epochs=10)
trainer.board.yscale = 'log'
trainer.fit(model, data)
```



The validation loss behaves as in Section 2.3: it falls by a factor of about 7 per epoch and levels off near the floor of 5×10^{-5}. The training curve lies on top of it. With three parameters and 1,000 examples, the model can fit almost none of the noise in its training labels, so it does as well on new examples as on the old ones. Has anything changed besides the organization of the code? The final validation loss and the parameter errors answer that.

```
w_error = d2l.numpy(d2l.reshape(model.w, data.w.shape) - data.w)
b_error = d2l.numpy(model.b - data.b)
print(f"validation loss: {trainer.history['val_loss'][-1]:.1e}")
print('error in w:', ', '.join(f'{e:.1e}' for e in w_error))
print('error in b:', ', '.join(f'{e:.1e}' for e in b_error))
```

```
validation loss: 5.1e-05
error in w: 4.3e-04, 3.0e-04
error in b: -6.7e-04
```

Nothing has. The validation loss is at the floor, and the parameters are within about 10^{-3} of the true ones, typically a few times 10^{-4}, as in Section 2.3. For a single model the classes would not be worth their extra lines. The next section is the first to need them: it replaces the model's parameters, loss, and optimizer with framework components and trains the result with this trainer, unchanged.

2.4.6 Summary

Every model in this book is trained by the same loop, so we write it once, as a `Trainer`, and keep apart what changes between models. A `Module` holds the parameters, the forward pass, the loss, and the optimizer; a `DataModule` serves training and validation data from loaders that can be iterated in every epoch. A model's steps return minibatch means and do nothing else, so any loop can train or evaluate the model. The trainer averages these means over the examples of each epoch, weighting each minibatch by its size, and keeps the results in `trainer.history`. Rewritten in this form, the linear regression model of Section 2.3 reaches the loss floor and recovers the true parameters as before, and a change to training is now made in one place.

2.4.7 Exercises

1. code **Methods added later.** Register a second method `do` on class `A` with a different body, predict which version `a.do()` runs, and check. Then attach a function `greet(self)` to the instance instead, with `setattr(a, 'greet', greet)`, call `a.greet()`, and explain the error.
2. **Optimizer and model.** The `Module` class keeps the optimizer in `configure_optimizers`, a *method of the model*, rather than passing it as an argument to `Trainer`. State the advantages of this design choice, and describe a training setup in which putting the optimizer on the model becomes awkward.
3. **Reading the training curves.** The trainer draws the mean training loss of each epoch at the middle of the epoch and the validation loss at its end.
    1. Early in training, the training curve often lies above the validation curve, although both sets of examples come from the same distribution. Explain why from the way the two means are computed.
    2. Suppose `fit` computed the training loss like the validation loss, with the parameters at the end of each epoch. What would this cost per epoch, and what would the gap between the two curves then measure?
4. code **Early stopping.** Training for a fixed number of epochs may stop too early or too late. Write a subclass of `d2l.Trainer` whose `fit` ends training once the validation loss

has not decreased for `patience` consecutive epochs, deciding from `self.history` alone. Explain why no model or dataset class has to change. Test your trainer on the synthetic regression data with a large `max_epochs`.

5. code **Test split.** Write a subclass of `SyntheticRegressionData` whose constructor also draws a test set from the same distribution and whose method `test_dataloader` serves it. Train a model on the original data, and compute its mean loss on the test set once, after the last epoch, with the model's `validation_step`. Explain why the test set is evaluated once at the end rather than after every epoch like the validation data, and what would go wrong if the epoch with the best test loss were reported. Section 2.6 returns to this question.

6. code **Planck's law.** ☒ The same `Module` and `Trainer` fit any model whose loss can be differentiated, not only a linear one. Use Planck's law to determine the temperature of a black body from measurements of its spectral radiance at several wavelengths. The spectral radiance is

$$B(\lambda, T) = \frac{2hc^2}{\lambda^5} \cdot \left(\exp \frac{hc}{\lambda k T} - 1 \right)^{-1}, \tag{2.23}$$

where λ is the wavelength, T is the temperature, c is the speed of light, h is Planck's constant, and k is the Boltzmann constant. Generate measurements at a known T, then write a `d2l.Module` subclass whose `forward` computes $B(\lambda, T)$ with T as its only parameter and fit it with `d2l.Trainer`. Choose units in which B and T are of order one before fitting.

7. **A partial last minibatch.** With 1,000 examples per split and `batch_size=32`, each loader ends a pass with a partial minibatch of 8 examples. `DataLoader` discards it when given `drop_last=True`. The training loader reshuffles every epoch, whereas the validation loader reads in a fixed order. For each loader, state which examples a discarded partial minibatch leaves out over many epochs, and decide for which loader discarding is acceptable.

8. code **On-the-fly generation.** Write a `DataModule` subclass that stores no **X** or **y**. Its training loader instead draws every minibatch fresh from the data-generating process when the minibatch is requested.

    1. A stream of fresh minibatches has no natural end, yet the `Trainer` ends an epoch only when a pass over the training loader ends. Decide how many minibatches one epoch yields, make every pass stop there, and make `len` return that number.
    2. Decide whether the validation loader should also draw fresh examples on every pass, and implement your choice. Justify it by what the validation loss is used for.

2.5 Concise Implementation

In Section 2.3 we wrote every part of linear regression by hand, from the initialization of the parameters to the update of minibatch stochastic gradient descent. In Section 2.4 we arranged the same parts into classes and trained the model once more through `d2l.Trainer`. Both steps were feasible because the model has three parameters. The models of later chapters are far larger. The image classifier ResNet-50 of Section 7.4, for instance, has 25.6 mil-

lion parameters in 161 tensors, and it is trained with update rules that keep a running average of past gradients for every parameter. Writing and testing all of this by hand for every new model would take most of our effort. Worse, a mistake in any part rarely raises an error. The model still trains, only less well.

Deep learning frameworks such as MXNet (Chen et al. 2015), JAX (Frostig et al. 2018) with its libraries Flax and Optax, PyTorch (Paszke et al. 2019), and TensorFlow (Abadi et al. 2016) therefore provide these parts ready-made. Their layers, losses, and optimizers are tested by many users and run on code tuned for each kind of hardware, and one loss or optimizer can be swapped for another by changing a line. This section swaps our hand-written parts for the framework's. A fully connected layer holds the weights and the bias and computes the predictions, a built-in loss computes the squared error, and an optimizer object applies the update. The `Trainer` runs unchanged, because it calls only the methods that every `Module` has. Only the scale of the loss needs some care.

2.5.1 The Fully Connected Layer

Linear regression is a single fully connected layer (Figure 2.6), and every framework provides that layer. For a minibatch with design matrix $\mathbf{X}$ it computes the predictions $\mathbf{X}\mathbf{w} + b$ that the `forward` method of our scratch model wrote out, and it creates and stores its own parameters. All that is left for us to choose is the number of outputs, here 1, and how the parameters are initialized. As in the scratch model, the weights are drawn from a normal distribution with mean 0 and standard deviation 0.01, and the bias starts at 0.

PyTorch's `nn.LazyLinear(1)` takes the number d of inputs from the first input it receives and has no parameters until then, which spares us counting the inputs of every layer in deeper networks. The method `weight_init` states our initialization for any linear layer; from Section 5.7 on, the trainer applies it after the first forward pass.

```
class LinearRegression(d2l.Module):
    """Linear regression built from framework components."""
    def __init__(self, lr):
        super().__init__()
        self.save_hyperparameters()
        self.net = nn.LazyLinear(1)

    def weight_init(self, module):
        if isinstance(module, nn.Linear):
            nn.init.normal_(module.weight, std=0.01)
            nn.init.zeros_(module.bias)
```

The model needs no `forward` method of its own. It inherits the `forward` of `Module` (Section 2.4), which passes its input to `self.net`, here the layer.

2.5.2 The Loss Function

Our scratch loss was half the squared error, averaged over the minibatch $\mathcal{B}$. Most frameworks provide the *mean squared error* instead,

$$\frac{1}{|\mathcal{B}|} \sum_{i \in \mathcal{B}} \left(\hat{y}^{(i)} - y^{(i)}\right)^2, \tag{2.24}$$

which has no factor $\frac{1}{2}$. It is the usual measure of prediction quality in regression: its square root is the typical size of an error, in the units of the label. The two losses have the same minimizer, so what we fit does not change. The steps do. A learning rate has a meaning only together with the scale of its loss (Section 2.1.2.3), and the framework loss is twice ours. To repeat the steps that the scratch model took at $\eta = 0.03$, this model must therefore be trained at $\eta = 0.015$.

The loss also decides what becomes of the silent error of Section 2.3, in which labels of shape $(32,)$ are broadcast against predictions of shape $(32, 1)$ into a 32×32 matrix of differences.

`nn.MSELoss` averages the squared errors over all entries of its inputs, here the examples of the minibatch. If labels and predictions differ in shape, it warns and then averages over the 32×32 matrix of differences anyway, so training proceeds on the wrong loss.

```
@d2l.add_to_class(LinearRegression)
def loss(self, y_hat, y):
    fn = nn.MSELoss()
    return fn(y_hat, y)
```

2.5.3 The Optimizer

The optimizer replaces our `SGD` class of Section 2.4. It receives the parameters and the learning rate η, and at each step it subtracts η times the gradient of the minibatch loss from each parameter, the update we wrote by hand. The optimizer finds the parameters through the layers of the model, so a model with more layers needs no change here. Each framework offers the other common update rules behind the same interface, and replacing minibatch SGD by one of them changes only this one line.

`torch.optim.SGD` takes the parameters as `self.parameters()`, which collects the parameters of every layer of the model, here the weight and the bias of `net`.

```
@d2l.add_to_class(LinearRegression)
def configure_optimizers(self):
    return torch.optim.SGD(self.parameters(), self.lr)
```

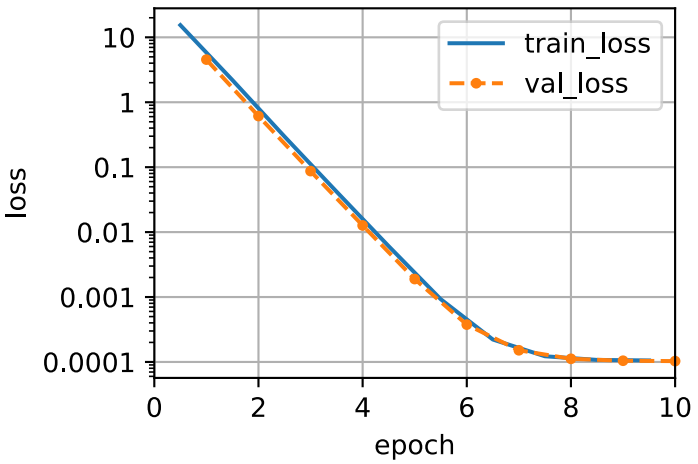
2.5.4 Training

We train on data drawn as before, from the generator of Section 2.2 with $\mathbf{w}^* = (2, -3.4)$, $b^* = 4.2$, and noise of standard deviation $\sigma = 0.01$: 1,000 training examples, served in minibatches of 32, and 1,000 validation examples. From the same initial parameters and on the same minibatches, the two models would compute the same updates (an exercise arranges this). Here the examples, their order, and the initial weights are all drawn afresh, so we should expect the behavior of the scratch run but not its digits.

Before training we call the model once on one example to create the parameters of the lazy layer, and `apply` then calls `weight_init` on every layer of `net`. Without this line the layer would keep PyTorch's default initialization, a uniform distribution whose width shrinks with the number of inputs.

We also put the trainer's plot, `trainer.board`, on a logarithmic scale, so that the loss stays visible after it has fallen by several orders of magnitude.

```
data = d2l.SyntheticRegressionData(w=d2l.tensor([2, -3.4]), b=4.2)
model = LinearRegression(lr=0.015)
model(data.X[:1])  # Creates the parameters of the lazy layer
model.net.apply(model.weight_init)
trainer = d2l.Trainer(max_epochs=10)
trainer.board.yscale = 'log'
trainer.fit(model, data)
```



The plot shows the mean training loss of each epoch and the validation loss after each epoch. Both curves fall along a straight line, by a factor of about 7 per epoch, and level off near 10^{-4} after about eight epochs. We have seen this plot before. The slope is the one that Section 2.3 predicted and measured, as it must be if the steps are the same. Only the height differs, since every loss is now doubled. In particular the floor, the loss of the true parameters, is the noise variance $\sigma^2 = 10^{-4}$ and no longer $\sigma^2/2$. To compare the run with this number we read the final validation loss from `trainer.history`.

```
print(f"validation loss: {trainer.history['val_loss'][-1]:.2e}")
```

```
validation loss: 1.03e-04
```

The final validation loss lies within about 15% of $\sigma^2 = 10^{-4}$.

What about the parameters themselves? The scratch model kept them as `self.w` and `self.b`. Here they belong to the layer, and `get_w_b` reads them out of `net`. A layer stores its weight as a matrix (a $1 \times d$ row in Gluon and PyTorch, a $d \times 1$ column called the `kernel` in Keras and Flax), so the comparison below first reshapes it to the shape of $\mathbf{w}^*$.

```
@d2l.add_to_class(LinearRegression)
def get_w_b(self):
    return self.net.weight.detach(), self.net.bias.detach()
```

```
w, b = model.get_w_b()
w_error = d2l.numpy(d2l.reshape(w, data.w.shape) - data.w)
b_error = d2l.numpy(b - data.b)
print('error in w:', ', '.join(f'{e:.1e}' for e in w_error))
print('error in b:', ', '.join(f'{e:.1e}' for e in b_error))
```

```
error in w: -6.0e-04, 7.6e-04
error in b: -6.9e-04
```

The parameter errors $\widehat{\mathbf{w}}-\mathbf{w}^*$ and $\widehat{b}-b^*$ are typically a few times 10^{-4} and rarely exceed 10^{-3}. The scratch model left errors of the same size, and they have the same sources (Section 2.3), chiefly the noise in the 1,000 training labels and what ten epochs leave of the initial error. Neither depends on who wrote the layer.

The rest of the book builds its models this way, from the layers, losses, and optimizers of the framework, and writes a part by hand only to show how it works or when the framework does not provide it. One habit of this chapter cannot be kept, however: judging a trained model by its distance from parameters that we chose ourselves. With real data there are no true parameters to compare with, and what matters is how well a model predicts examples it has not seen, the question of Section 2.6.

2.5.5 Summary

Deep learning frameworks provide tested and fast versions of the parts we wrote by hand for linear regression. A fully connected layer holds the weights and the bias and computes the predictions, a loss function computes the mean squared error, and an optimizer applies the SGD update to every parameter of the model. Replacing each hand-written part by its framework counterpart leaves `Module`, `DataModule`, and `Trainer` unchanged. A framework loss may differ from ours by a constant factor, and the learning rate must then be rescaled to match. Writing a part by hand remains the way to understand it, and the way to build one that the framework lacks.

2.5.6 Exercises

1. code **Loss scaling.** The framework loss in `LinearRegression.loss` is twice the loss of `LinearRegressionScratch`.
    1. For one minibatch $\mathcal{B}$ of `SyntheticRegressionData`, compute the gradient of `LinearRegression.loss` with respect to the weights by automatic differentiation, and compare it entry by entry with the formula $\frac{2}{|\mathcal{B}|}\sum_{i\in\mathcal{B}}\mathbf{x}^{(i)}\left(\mathbf{w}^\top\mathbf{x}^{(i)}+b-y^{(i)}\right)$ at the same parameters.
    2. Give `LinearRegression` and `LinearRegressionScratch` the same initial parameters and the same minibatches, for example by setting `batch_size` to `num_train` so that every step uses the whole training set. Train the first at `lr=0.015` and the second at `lr=0.03`, confirm that the trained parameters agree, and explain why the two training-loss curves still differ.
2. code **A layer without a bias.** Build `LinearRegression` with a layer that has no bias (every framework's dense layer has an argument that turns the bias off), and train it as in this section.
    1. Before training, predict the value at which the validation loss levels off, from the expected loss Equation 2.22 doubled for the mean squared error. Then check

your prediction.
   2. Predict what the weights converge to and how far they end from $\mathbf{w}^*$. Since the features have mean zero, the missing bias acts on the weights like label noise of standard deviation $b^* = 4.2$; it enters both the estimation error of Section 2.2.2 and the noise ball of minibatch SGD (Section 2.1). Compare your prediction with the errors of the trained weights.
3. code **Several outputs.** Change the output width of the layer in `LinearRegression` from 1 to 2. After constructing `data`, append to `data.y` a second label column $y_2 = \mathbf{x}^\top \mathbf{w}_2^* + b_2^* + \epsilon_2$, computed from `data.X` with generating parameters of your choice and noise of the same size.
   1. Show that the squared loss summed over the two outputs splits into two independent linear regressions, one per output. Conclude that the parameters learned for one output do not depend on the labels of the other.
   2. The framework losses average over every entry of the $|\mathcal{B}| \times 2$ matrix of errors of a minibatch $\mathcal{B}$, not only over its examples. State by which factor the gradient of each output's parameters shrinks relative to the single-output model, and which learning rate reproduces the single-output training run. Confirm this by training both models from the same initial parameters on the same minibatches.
4. code **Feature scale and step size.** Multiply the first column of `data.X` by 100 after constructing `data`, leaving `data.y` unchanged, so that the generating weight of that feature becomes 0.02. Recall from exercise 4 that gradient descent diverges once η exceeds $2/\mu_{\max}$, where $\mu_{\max}$ is the largest eigenvalue of the Hessian $\mathbf{H}$ of the loss.
   1. Compute $\mathbf{H}$ for `LinearRegression.loss` and estimate its eigenvalues before and after the rescaling.
   2. Predict the largest learning rate for which training does not diverge, before and after the rescaling, and check both predictions by training at learning rates around them.
   3. At a learning rate just below the new limit, predict how many updates the second weight needs to reduce its error by a factor of 100, and compare the prediction with a measurement. Then standardize the rescaled feature as in Section 1.2, and show that `lr=0.015` works again.
5. code **Optimization versus estimation error.** ⊠ Train `LinearRegression` with `lr=0.015` and `max_epochs=10` on `SyntheticRegressionData` with `num_train` set to each of $5, 50, 500, 5{,}000$ and `batch_size=min(32, num_train)`. For each run, compare the distance $\|\hat{\mathbf{w}} - \mathbf{w}^*\|_2$ of the trained weights from the generating weights $\mathbf{w}^*$, stored in `data.w`, with the same distance for the least-squares solution on the same training data, computed as in the recovery subsection (Section 2.2.2).
   1. Count the parameter updates in each run, and explain the ratio of the two errors at each `num_train`.
   2. Find the number of epochs after which the run with `num_train=5` comes within a factor of two of its least-squares error.

2.6 Generalization

The last three sections trained the same linear model, with two weights and a bias, on 1,000 synthetic examples, and each time it recovered the parameters we had chosen almost exactly. There, fitting the training data was the whole task, because we had generated the data ourselves from a known model. In practice we fit a model to the examples we have in order to predict the labels of examples we do not have yet, such as the sale price of a house that

comes on the market next month. Fitting the data is not enough for that. Given 20 measurements at distinct inputs, a polynomial of degree 19 passes through every one of them exactly, yet nothing guarantees that it predicts a 21st measurement well. A model that reproduces its training labels without capturing the pattern behind them has *memorized* them, like a student who learns last year's exam answers by heart and then faces new questions. What we want is *generalization*: good predictions on new examples drawn from the same population as the training data.

Every model is fit to a finite sample, so the danger of memorization is always present. Some medical studies can enroll only a few thousand patients, and studies of rare diseases may have only hundreds. The largest public collections of labeled photographs, such as ImageNet (Deng et al. 2009), contain millions of images. Even that is a vanishing fraction of all possible megapixel images, so almost every input a model meets after training is new.

How can we tell, from a finite sample, whether a model will generalize? We first define the error on new data and estimate it with examples held out from training. An experiment with polynomials then shows how this error depends on the flexibility of the model and on the amount of data. Finally, we use held-out data to choose among models without deceiving ourselves.

2.6.1 Training Error and Generalization Error

To say what "new examples from the same population" means, we need an assumption that ties the training data to the data the model will see later. The standard assumption is that all examples, those we train on and those we later predict, are drawn *independently* from one *identical* distribution $P(X, Y)$ over inputs and labels. This is the *IID assumption*. It is a bold one, and it often fails in practice. A model trained on photographs taken in daylight may be used at night, and a model trained on last year's purchases is applied to this year's. We cannot do without it, though: unless something relates the training distribution to the distribution at deployment, data from the first says nothing about performance on the second. Later chapters examine such shifts; here we assume IID data.

Under this assumption a model f has two errors. Its *training error*, also called the *empirical risk*, is the average loss over the n training examples,

$$R_{\text{emp}}(f) = \frac{1}{n} \sum_{i=1}^{n} l(f(\mathbf{x}^{(i)}), y^{(i)}), \tag{2.25}$$

where $l(\hat{y}, y)$ is the loss of predicting $\hat{y}$ when the label is y, such as the squared loss of Section 2.1. Its *generalization error*, also called the *risk*, is the expected loss on a new example $(\mathbf{x}, y)$ drawn from P:

$$R(f) = \mathbb{E}[l(f(\mathbf{x}), y)]. \tag{2.26}$$

The training error averages the loss over the sample we have. The generalization error averages it over the whole population, which we would see only by applying the model to an endless stream of new examples. Their difference $R - R_{\text{emp}}$ is the *generalization gap*. It is R that we care about. The gap tells us how far the training error, the only one of the two that we can compute, misleads us about it.

Unfortunately we cannot compute R, since we do not know P. But we can estimate it. To this end we hold out n' examples that play no part in training, and evaluate the formula for the training error on them instead. Since the model was chosen without them, their losses are n' independent draws of a random variable whose mean is R. Their average is therefore an unbiased estimate of R, and like any average of n' independent draws, it has the standard error $\sigma_l/\sqrt{n'}$, where σ_l is the standard deviation of the loss on one example. Four times as many held-out examples halve the error of the estimate. The validation loss that we have plotted since Section 2.3 is such an estimate. Below we call the held-out examples a *test set*. Test and validation sets need to be told apart only once we choose among models (Section 2.6.3).

The training error has no such guarantee. The training examples were used to choose the model, so their losses are not independent of it, and the model looks better on them than it is. An extreme case shows how large the effect can be. A model that stores the training set in a table and answers each training input with its stored label has zero training error, yet it has nothing to say about an input it has not seen. In general, the training error is an optimistic estimate of R. How optimistic? For our linear model, hardly at all: in Section 2.4 its loss on 1,000 held-out examples tracked its training loss closely, since a model with three parameters fit to 1,000 examples has little room to memorize. The rest of this section asks when the gap is large, and what to do about it.

2.6.2 Underfitting and Overfitting

How large the gap is depends on how flexible the model is compared with the amount of data. It is instructive to study this in an experiment in which we control both the true function and the flexibility of the model.

2.6.2.1 Polynomial Curve Fitting

Consider a single input x and a real-valued label y. A polynomial of degree d predicts

$$\hat{y} = \sum_{j=0}^{d} w_j x^j = w_0 + w_1 x + w_2 x^2 + \cdots + w_d x^d.$$

This is linear regression with the powers $x, x^2, \ldots, x^d$ as features: the weights are $w_1, \ldots, w_d$, and w_0 plays the role of the bias, since it multiplies the constant feature $x^0 = 1$. The degree controls the flexibility. A polynomial of higher degree contains every polynomial of lower degree (set the extra weights to zero), so on the same training set the best fit of a higher degree never has a larger training error. When the n training inputs are distinct, degree $n-1$ has as many weights as there are examples and fits every label exactly (exercise 3).

We generate the data from a known cubic,

$$y = f^*(x) + \epsilon \quad \text{with} \quad f^*(x) = 5 + 1.2x - 3.4x^2 + 5.6x^3 \quad \text{and} \quad \epsilon \sim \mathcal{N}(0, \sigma^2), \quad (2.27)$$

where $\sigma = 0.1$ and the inputs x are drawn uniformly from $[-1, 1]$. The noise ϵ is the part of the label that no model can predict, so its variance σ^2 is the error that no model can beat.

We know this floor only because we generated the data ourselves; real data do not reveal it. We draw only 20 training examples, so that high degrees have room to overfit, and 1,000 test examples, enough to estimate the generalization error of the models that fit well to within a few percent.

The experiment needs no framework: NumPy draws the data and solves the least-squares problems. `make_data` draws n inputs and labels them with the noisy cubic.

```
def target(x):
    """The cubic f* that generates the labels."""
    return 5 + 1.2 * x - 3.4 * x**2 + 5.6 * x**3

def make_data(rng, n, sigma):
    """Draw n inputs uniformly from [-1, 1] and label them by the noisy cubic."""
    x = rng.uniform(-1, 1, n)
    return x, target(x) + rng.normal(0, sigma, n)

sigma = 0.1                                # standard deviation of the noise
rng = np.random.default_rng()              # a new random draw on every run
x_train, y_train = make_data(rng, 20, sigma)  # few, so high degrees overfit
x_test, y_test = make_data(rng, 1000, sigma)
```

Fitting a polynomial of degree d is least squares on the design matrix whose columns are $1, x, \dots, x^d$. As in Section 2.1, we let the library routine `lstsq` compute the minimizer directly instead of running gradient descent. We report the mean squared error, without the factor $\frac{1}{2}$ of the training loss, so that a model that predicts f^* exactly scores about $\sigma^2 = 0.01$ on new data.

```
def features(x, d):
    """Design matrix whose columns are the powers x^0, x^1, ..., x^d."""
    return x.reshape(-1, 1) ** np.arange(d + 1)

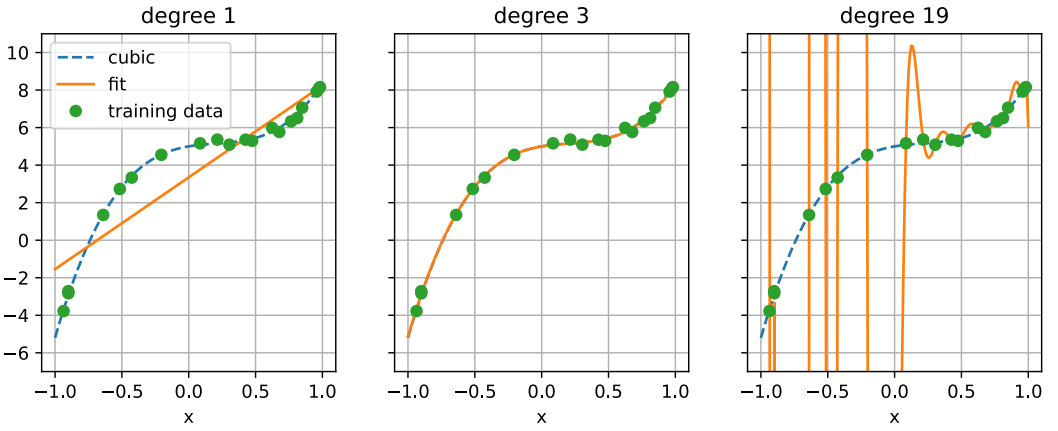
def fit_degree(x, y, d):
    """Least-squares weights of the degree-d polynomial fitted to (x, y)."""
    return np.linalg.lstsq(features(x, d), y, rcond=None)[0]

def mse(w, x, y):
    """Mean squared error of the polynomial with weights w on (x, y)."""
    return ((features(x, len(w) - 1) @ w - y) ** 2).mean()
```

How do fits of degree 1, 3, and 19 compare? We plot each one against the training data and the true cubic.

```
xs = np.linspace(-1, 1, 200)
_, axes = d2l.plt.subplots(1, 3, figsize=(10, 3.4), sharey=True)
for ax, d in zip(axes, (1, 3, 19)):
    w = fit_degree(x_train, y_train, d)
    d2l.plot([xs, xs, x_train], [target(xs), features(xs, d) @ w, y_train],
             xlabel='x', ylim=(-7, 11), fmts=('--', '-', 'o'),
             legend=['cubic', 'fit', 'training data'] if d == 1 else None,
             axes=ax)
```

```
    ax.set_title(f'degree {d}')
```



The three panels show the two ways in which a model can fail. The straight line (degree 1) cannot bend to follow the cubic, so it misses training points and new points alike. This is *underfitting*. The cubic fit (degree 3) is hard to tell apart from the truth. The polynomial of degree 19 passes through all 20 training points but swings far away from the cubic between them. This is *overfitting*: the model fits the particular sample, noise included, at the expense of the pattern. The training and test errors, and the largest weight of each fit, quantify what the panels show.

```
for d in (1, 3, 19):
    w = fit_degree(x_train, y_train, d)
    print(f'degree {d:2d}: train {mse(w, x_train, y_train):.2g}, '
          f'test {mse(w, x_test, y_test):.2g}, '
          f'largest |weight| {np.abs(w).max():.2g}')
```

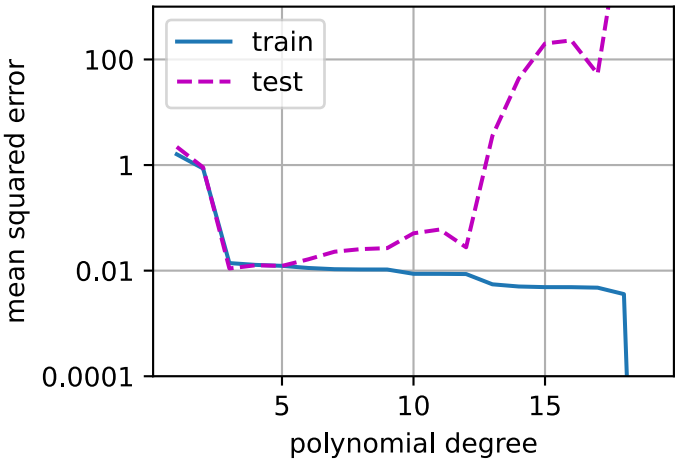
```
degree  1: train 1.6, test 2.2, largest |weight| 4.9
degree  3: train 0.014, test 0.011, largest |weight| 5.5
degree 19: train 3.4e-17, test 5.5e+07, largest |weight| 7.8e+07
```

Every run draws new data, so the numbers change from run to run, but the pattern does not. Degree 1 has errors far above the noise level on both splits; its test error exceeds 1, more than a hundred times $\sigma^2 = 0.01$. Degree 3 brings both errors down to the order of the noise level: it predicts everything except the noise. Degree 19 has a training error far below the noise level and a test error many orders of magnitude larger than that of degree 3. The weights show why. The largest weight of the cubic fit is close to the true coefficient 5.6. The weights of the fit of degree 19 are larger by many orders of magnitude, and their huge positive and negative terms cancel at the 20 training inputs and almost nowhere else.

To see how the errors change between these extremes, we sweep the degree from 1 to 19. The vertical axis is logarithmic, and errors beyond its range are cut off.

```
degrees = list(range(1, 20))
weights = [fit_degree(x_train, y_train, d) for d in degrees]
train_err = [mse(w, x_train, y_train) for w in weights]
```

```
test_err = [mse(w, x_test, y_test) for w in weights]
d2l.plot(degrees, [train_err, test_err], xlabel='polynomial degree',
         ylabel='mean squared error', legend=['train', 'test'],
         yscale='log', ylim=(1e-4, 1e3))
```



Training error falls with every added degree, as it must, and at degree 19 it lies far below the noise level, usually at zero up to rounding. Test error behaves quite differently. It falls steeply until degree 3, is lowest at degree 3 or a little above it, and then climbs by orders of magnitude. The curve shows both failures at once: to the left of degree 3 the model is too rigid to represent the cubic, and to the right its extra flexibility goes into fitting the noise. Comparing the two errors tells us which failure we face. Underfitting shows as a training error that stays high: the model cannot fit even the data it was trained on. Overfitting shows as a test error far above a low training error. A large gap is not a defect in itself, though; what we care about is a low test error, and the best deep models often fit their training data far better than held-out data.

2.6.2.2 Bias and Variance

From degree 3 on, every model class contains the true cubic. Why, then, does test error rise again? The fitted polynomial depends on the particular noise in the 20 training labels. If we repeated the experiment with the same inputs and fresh noise, we would get a different fit, and the more flexible the model, the more the fits would differ. To make this precise, fix a test input x with label $y = f^*(x) + \epsilon$, and let $\hat{f}$ be the model fitted to a random draw of the training noise. Taking the expectation over both the training noise and the test noise ϵ, the expected squared error splits into three parts:

$$\mathbb{E}\left[\left(y - \hat{f}(x)\right)^2\right] = \underbrace{\left(f^*(x) - \mathbb{E}[\hat{f}(x)]\right)^2}_{\text{bias}^2} + \underbrace{\mathrm{Var}\left(\hat{f}(x)\right)}_{\text{variance}} + \sigma^2. \tag{2.28}$$

The first term, the squared *bias*, is the squared distance between the average fit and the truth: a systematic error that remains however often we repeat the experiment. This bias, an error of the fitted function, is unrelated to the bias b of a linear model. The second, the

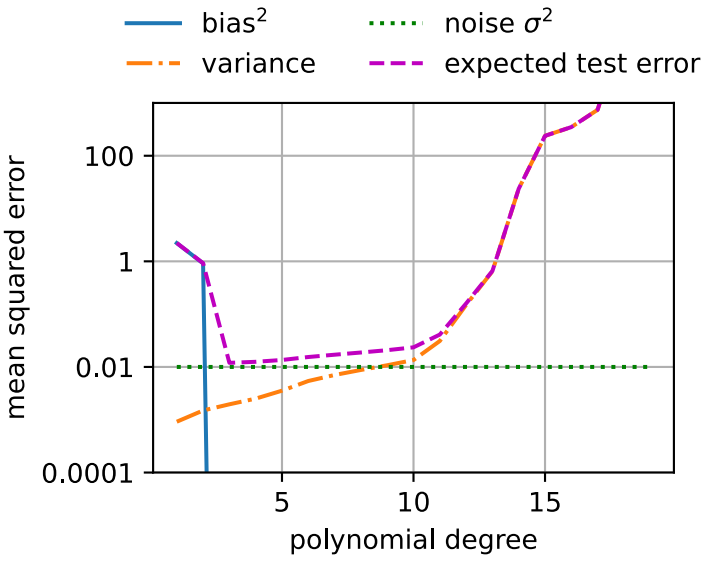
variance, measures how much the fit changes from one draw of the training noise to the next. The third is the noise in the test label. To see why the error splits this way, name the three parts of the prediction error: the test noise ϵ, the bias $\text{bias}(x) = f^*(x) - \mathbb{E}[\hat{f}(x)]$, and the fluctuation $v = \mathbb{E}[\hat{f}(x)] - \hat{f}(x)$ of the fit around its average. Then $y - \hat{f}(x) = \epsilon + \text{bias}(x) + v$, and

$$\mathbb{E}\left[\left(\epsilon + \text{bias}(x) + v\right)^2\right] = \mathbb{E}[\epsilon^2] + \text{bias}(x)^2 + \mathbb{E}[v^2] = \sigma^2 + \text{bias}(x)^2 + \text{Var}(\hat{f}(x)),$$

because every cross term has an expectation of zero: $\text{bias}(x)$ is a constant, ϵ and v have mean zero, and ϵ is independent of v, which depends only on the training noise. The mathematics appendix derives the decomposition for general estimators (Section 36.5).

A decomposition is more convincing when its terms can be measured, and in our experiment they can. The key observation is that the least-squares weights $\hat{\mathbf{w}} = (\mathbf{X}^\top \mathbf{X})^{-1} \mathbf{X}^\top \mathbf{y}$ are a linear function of the labels, and the noise has mean zero. The average of the fits over many draws of the noise is therefore the fit to the noiseless labels $f^*(x^{(i)})$, and its error against f^* on the test inputs is the squared bias. For the variance, we simply refit on 200 fresh draws of the training noise and measure how much the predictions at each test input spread. Both terms are averaged over the test inputs.

```
noise = rng.normal(0, sigma, (len(x_train), 200))   # 200 draws of the noise
bias2, var = [], []
for d in degrees:
    w_avg = fit_degree(x_train, target(x_train), d)              # average fit
    W = fit_degree(x_train, target(x_train)[:, None] + noise, d)  # fit per draw
    bias2.append(mse(w_avg, x_test, target(x_test)))
    var.append((features(x_test, d) @ W).var(axis=1).mean())
d2l.plot(degrees, [bias2, var, [sigma**2] * len(degrees),
                   np.add(bias2, var) + sigma**2],
         xlabel='polynomial degree', ylabel='mean squared error',
         fmts=('-', 'C1-.', 'g:', 'm--'), yscale='log', ylim=(1e-4, 1e3))
d2l.plt.legend(['bias$^2$', 'variance', r'noise $\sigma^2$',
                'expected test error'], ncol=2, frameon=False,
               loc='lower center', bbox_to_anchor=(0.5, 1));  # above the plot
```



The squared bias is large for degrees 1 and 2 and drops to zero, below the bottom of the plot, at degree 3, where the model class first contains the cubic. The variance grows with every added degree. At low degrees it is roughly $\sigma^2(d+1)/n$, the noise variance times the number of weights per training example, and small compared with the noise; exercise 5 shows that this holds exactly when the variance is averaged over the n training inputs instead of the test inputs. At high degrees the variance exceeds the noise by orders of magnitude. The sum of the three terms, the expected test error, is smallest at degree 3. This answers our question. Underfitting is error from bias, and overfitting is error from variance.

Nothing in this trade-off is special to polynomials. It appears in many model families, as Figure 2.8 sketches. Training error falls steadily. Test error falls while the bias shrinks and rises once the variance dominates, so it is smallest at an intermediate complexity, and it never drops below the noise level.

2.6.2.3 Model Complexity

The degree of a polynomial measures complexity by counting parameters, which is convenient but rather crude. What makes a class of models complex in general is its ability to fit many different labelings of the same inputs, which we call its *capacity*. Consider a class so flexible that for any n inputs it can fit any labels perfectly, even labels assigned at random. Polynomials of degree 19 on our 20 distinct inputs are such a class. A perfect fit to the training data then tells us nothing, because the class would have fit any other labels just as well (Vapnik et al. 1994). The philosopher Karl Popper made the same point about scientific theories: a theory must rule out some possible observations, and a theory that can explain any outcome gains no support from the outcome it explains (Popper 2005). Conversely, if a class cannot fit arbitrary labels and the sample is large compared with its capacity, a low training error is evidence that the model has found a real pattern. Learning theory turns this into bounds on the generalization gap, which the mathematics appendix develops (Section 36.10).

The number of parameters is not all that matters. Their size matters too: the polynomial of degree 19 fit the training points only by using enormous weights. A polynomial whose

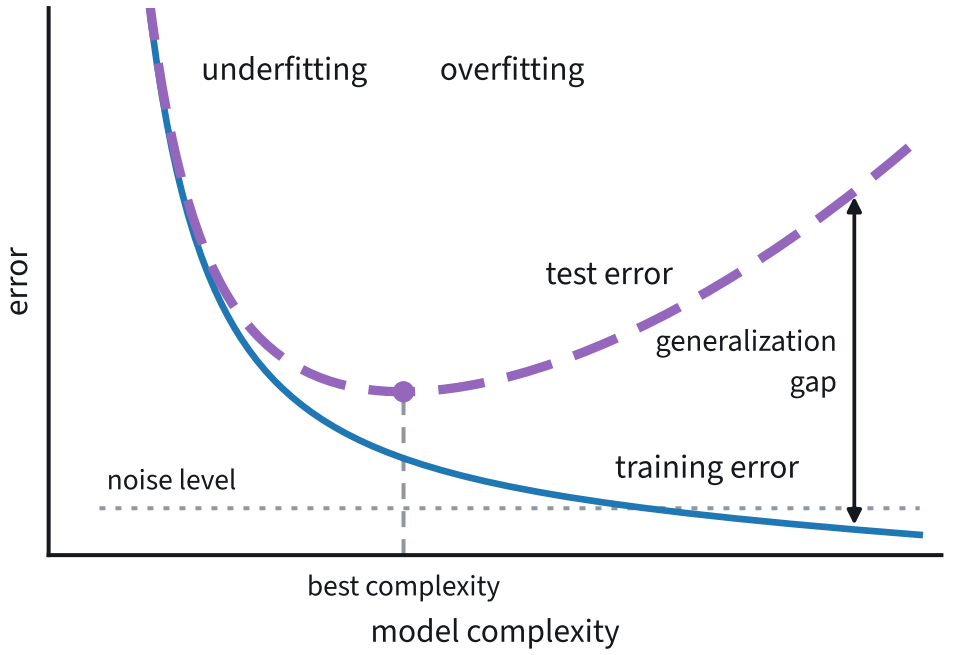


Figure 2.8: Training and test error against model complexity. To the left of the best complexity the model underfits and both errors are high; to the right it overfits, its training error falls below the noise level, and the generalization gap widens.

weights must stay small cannot swing so wildly between data points, whatever its degree. Limiting the size of the weights is the idea behind weight decay, the subject of the next section.

For deep neural networks, counting parameters fails even more clearly. Large networks can fit randomly assigned labels on standard image datasets, and yet, surprisingly, trained on the real labels they generalize well (Zhang et al. 2021). Their training error alone therefore says little about how well they generalize, and we rely on held-out data to measure it. Section 4.5 returns to this puzzle once we have built such networks.

2.6.2.4 Dataset Size

Complexity is relative to the amount of data. A polynomial of degree 19 has as many weights as our 20 training examples, so it can memorize them. With 1,000 examples it cannot, and the noise in individual labels averages out. We check this by fitting degrees 1, 3, and 19 to fresh training sets of increasing size and evaluating each fit on the same test set.

```
for n in (20, 50, 100, 1000):
    x, y = make_data(rng, n, sigma)          # a fresh training set of size n
    errs = [mse(fit_degree(x, y, d), x_test, y_test) for d in (1, 3, 19)]
    print(f'n = {n:4d}: test error of degrees 1, 3, 19:',
          ', '.join(f'{e:.2g}' for e in errs))
```

```
n =   20: test error of degrees 1, 3, 19: 2.3, 0.016, 5.8e+08
n =   50: test error of degrees 1, 3, 19: 2, 0.012, 0.14
n =  100: test error of degrees 1, 3, 19: 1.9, 0.011, 0.012
n = 1000: test error of degrees 1, 3, 19: 2, 0.011, 0.011
```

More data helps the overfitting model enormously. With 100 examples the test error of the polynomial of degree 19 has fallen by orders of magnitude, and with 1,000 it matches that of the cubic. The straight line hardly improves: with 1,000 examples its test error is still about 1.5 to 2. More data reduces the variance, but not the bias. For models like these, whose capacity grows with the number of weights, the rule is that the complexity of a model should grow with the amount of data, not ahead of it. Deep networks will complicate this rule (Section 4.5).

2.6.3 Model Selection

In the experiment we picked degree 3 by looking at the test error. That was a demonstration, not a method. In practice we choose among many candidate models, which may differ in architecture, in features, in the training objective, or in hyperparameters (settings chosen before training rather than learned) such as the learning rate and the number of epochs. Choosing among candidates is called *model selection*, and we must do it without the test set.

2.6.3.1 Validation Sets

Why not select on the test set? Because the test error of the selected model is then no longer an unbiased estimate of its generalization error. If we try 19 degrees and keep the one with the smallest test error, we keep the model whose test error came out lowest partly by luck, and its reported error is biased downward: we have overfit the test set. The bias grows with the number of choices made this way (Ong et al. 2005). Nor can we select on the training data, because the training error always favors the most flexible candidate. In our sweep it would pick degree 19.

The standard remedy is to split the data three ways. The *training set* fits the parameters of each candidate. The *validation set* estimates the generalization error of each candidate, and we keep the best. The test set is used once, at the end, to estimate the error of the chosen model. That is the ideal. In practice test sets are reused across many rounds of experiments and many papers, which invites the same bias. This book is no exception: the held-out data we report results on also serves to compare models, so unless stated otherwise, the accuracies we report are validation accuracies rather than test accuracies.

2.6.3.2 Cross-Validation

When data is scarce, holding out a validation set of useful size leaves too little for training: with our 20 examples, a validation set of 5 would take a quarter of the data. *K-fold cross-validation* uses every example for both purposes. It splits the training data into K non-overlapping parts, called *folds*. Each candidate model is trained K times, each time on $K-1$ folds, and evaluated on the remaining fold, as in Figure 2.9. The average of the K validation errors estimates the candidate's generalization error. Once a candidate is chosen, we train it again on all the training data.

The folds must be random subsets of the data. Our 20 training examples were drawn in random order, but real datasets often arrive sorted, by time, by label, or by the value of a feature, and folds cut from consecutive examples would then each cover only part of the

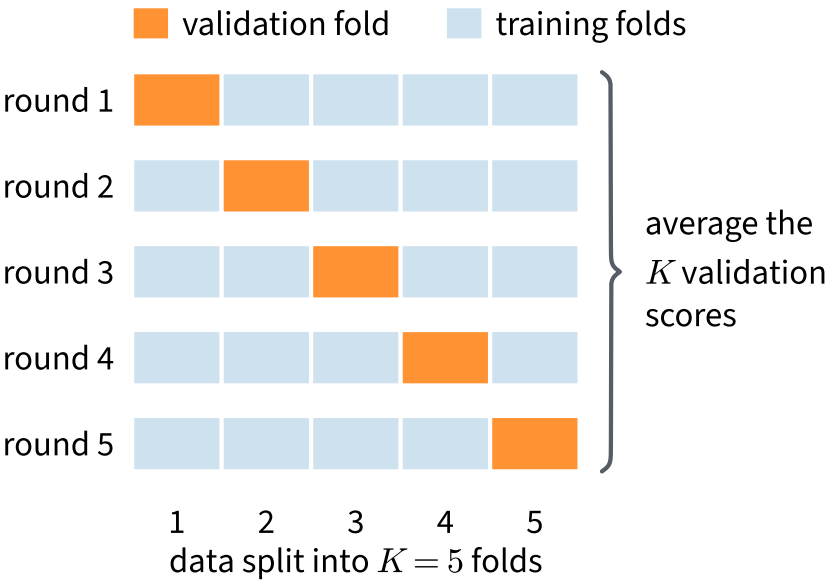


Figure 2.9: In K-fold cross-validation, each of the K folds serves once as the validation set (orange) while the model trains on the remaining folds (blue); the final estimate averages the K validation scores.

inputs. `kfold_splits` therefore divides a random permutation of the indices into K folds with `np.array_split`. The estimate also depends on which examples happen to share a fold, noticeably so with folds of four examples. We reduce this dependence by drawing several random splits, ten by default, and averaging the validation errors over all of their folds, which is called *repeated* cross-validation. Inside `cv_error`, `np.delete` removes the fold `f` from the training data.

```
def kfold_splits(rng, n, K=5, repeats=10):
    """Random partitions of the indices 0, ..., n-1 into K folds."""
    return [np.array_split(rng.permutation(n), K) for _ in range(repeats)]

def cv_error(x, y, d, splits):
    """Validation error of the degree-d fit, averaged over all folds."""
    return np.mean([mse(fit_degree(np.delete(x, f), np.delete(y, f), d),
                        x[f], y[f]) for folds in splits for f in folds])
```

We now select the degree using only the 20 training examples. Every degree is evaluated on the same splits, so that differences between degrees do not come from differences between splits. We also print the degree with the smallest test error, for comparison only; the selection never sees it.

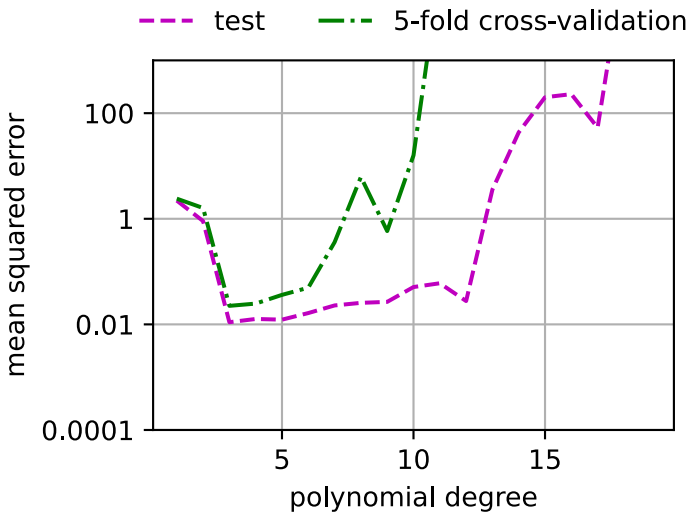
```
splits = kfold_splits(rng, len(x_train))  # the same folds for every degree
cv_err = [cv_error(x_train, y_train, d, splits) for d in degrees]
print('cross-validation picks degree:', degrees[np.argmin(cv_err)])
```

```
print('lowest test error at degree:  ', degrees[np.argmin(test_err)])
```

```
cross-validation picks degree: 3
lowest test error at degree:   3
```

How closely does the cross-validation error follow the test error across all degrees?

```
d2l.plot(degrees, [test_err, cv_err], xlabel='polynomial degree',
         ylabel='mean squared error', fmts=('m--', 'g-.'), yscale='log',
         ylim=(1e-4, 1e3))
d2l.plt.legend(['test', '5-fold cross-validation'], ncol=2, frameon=False,
               loc='lower center', bbox_to_anchor=(0.5, 1));  # above the plot
```



Cross-validation finds the bottom of the test error curve without touching the test set, though not every time. It usually picks degree 3, and otherwise a degree a little above it, whose test error is close to that of degree 3. Exercise 7 measures how often each choice occurs and what a wrong choice costs. Near the bottom the two curves agree to within a small factor, and the cross-validation estimate is the noisier of the two, because it rests on 20 validation examples instead of 1,000. At high degrees it usually rises earlier than the test error, and far higher. Each fold's model is fit to 16 points instead of 20, so at a given degree it has more weights per training point: degree 15 already passes through all 16 points, as degree 19 does through all 20. Some validation points also lie beyond the range of the 16 fitted points, near the ends of $[-1, 1]$, where a polynomial of high degree swings far from the data.

How should we choose K? Each fold's model trains on only a fraction $(K-1)/K$ of the data. If the error falls as the amount of training data grows, as it usually does, each fold's model is somewhat worse than the model we finally train on all the data, and the cross-validation estimate is pessimistic. Taking $K = n$, called *leave-one-out* cross-validation, all but removes this bias, but it requires n fits per candidate. Its n training sets are also nearly identical, so the fold errors are strongly correlated and their average tends to have a high variance. The usual compromise, $K = 5$ or $K = 10$, keeps the bias moderate at the cost of

5 to 10 fits (Kohavi 1995). Exercises 8 and 9 work out the bias of the K-fold estimate, and a shortcut that makes leave-one-out as cheap as a single fit for least squares.

Note, however, that the polynomial degree is a coarse control, however carefully we select it. Going from degree 3 to 4 adds a whole new feature, and with many input variables the number of polynomial features grows so fast that each step in degree is a large jump in capacity. Weight decay, which penalizes the enormous weights we met at degree 19, will let us adjust capacity continuously.

2.6.4 Summary

A model is useful only if it predicts well on new data from the distribution of its training data, and its training error alone cannot tell us whether it does. Under the IID assumption, the error on held-out data estimates the generalization error, with a standard error that shrinks like $1/\sqrt{n'}$ in the number n' of held-out examples. A model that is too rigid underfits: its training error is high, because of bias. A model that is too flexible for its data overfits: its training error is low and its test error high, because of variance. More data reduces the variance and lets us afford more flexible models, while validation sets and K-fold cross-validation choose the flexibility without touching the test set.

2.6.5 Exercises

1. **When IID fails.** Give three examples in which dependent observations make treating a dataset as IID inadvisable. For each, state whether a random train/validation split overestimates or underestimates the generalization error, and describe a split that avoids the problem.

2. **Two kinds of zero error.** Under which circumstances can the training error be driven to zero? Under which circumstances can the generalization error be zero? Compare the two sets of conditions, and explain the role of the noise term σ^2 in the decomposition of the expected test error, Equation 2.28.

3. **Exact polynomial regression.** Show that when the n training inputs $x^{(1)}, \dots, x^{(n)}$ are distinct, a polynomial of degree $n - 1$ fits the n labels exactly, and explain where distinctness is used. What does least squares return when two inputs coincide but their labels differ?

4. code **A better basis for degree 19.** The high powers x^{17} and x^{19} differ by less than 0.05 anywhere on $[-1, 1]$, so the columns of `features(x, 19)` are nearly parallel. The Chebyshev polynomials $T_0, \dots, T_{19}$ span the same functions but stay between -1 and 1 on the interval.

    1. Predict first: if we fit degree 19 to the 20 training points in the Chebyshev basis, which of the training error, the test error, and the largest weight will change, and by how much?
    2. Check your prediction over several runs: replace `features(x, 19)` by `np.polynomial.chebyshev.chebvander(x, 19)` for both the training and the test inputs, and compare the two design matrices' condition numbers (`np.linalg.cond`), the training and test errors, and the largest weights.
    3. Explain why the two fits are the same polynomial in exact arithmetic, and which of your numbers differ in float64 and why. Does the better basis rescue degree

19 on 20 points? What does your answer say about the size of the weights as a measure of complexity?

5. **Optimism of the training error.** ⊠ Consider least squares with a fixed $n \times p$ design matrix $\mathbf{X}$ of full column rank and labels $\mathbf{y} = \mathbf{X}\mathbf{w}^* + \epsilon$, whose noise terms are independent with mean zero and variance σ^2. The fitted predictions are $\hat{\mathbf{y}} = \mathbf{H}\mathbf{y}$ with $\mathbf{H} = \mathbf{X}(\mathbf{X}^\top\mathbf{X})^{-1}\mathbf{X}^\top$.
   1. Show that the expected training error is $\mathbb{E}[\|\mathbf{y} - \hat{\mathbf{y}}\|^2]/n = \sigma^2(n-p)/n$. Hint: $\mathbf{y} - \hat{\mathbf{y}} = (\mathbf{I} - \mathbf{H})\epsilon$, and the trace of $\mathbf{I} - \mathbf{H}$ is $n - p$.
   2. Draw fresh labels $\mathbf{y}' = \mathbf{X}\mathbf{w}^* + \epsilon'$ at the same inputs. Show that the expected error of $\hat{\mathbf{y}}$ on them is $\sigma^2(n+p)/n$, so that the training error underestimates it by $2\sigma^2 p/n$. Show that $\sigma^2 p/n$ is the variance of the predictions, averaged over the n inputs.
   3. Evaluate both expressions for the cubic fit of Section 2.6.2.1 ($n = 20$, $p = 4$, $\sigma = 0.1$) and compare them with the training and test errors of degree 3 over several runs of the experiment. Explain the differences: how much does the training error of a single run spread around its expectation, and why does the formula of (b), which holds at the training inputs, not apply exactly to the test inputs?

6. code **Data size and model complexity.** Regenerate the polynomial data of Section 2.6.2.1 and report every test error as the median over 20 runs, because a single draw fluctuates by about as much as the differences measured here.
   1. Your manager asks whether collecting more data would improve a model that performs poorly, before paying for any. Draw 100 training examples, fit degrees 1, 3, and 10 on the first 20, 40, 60, 80, and 100 of them, and plot the test error of each degree against the training-set size. Explain how the slope of each curve at 100 examples supports or undermines a request for more data.
   2. Sweep the degree from 1 to 29 with 10, 40, and 100 training examples. For each size, report the first degree above 3 at which the test error exceeds twice its value at degree 3. Relate the trend to the rule that the complexity of a model should grow with the amount of data, not ahead of it.

7. code **Reliability of cross-validation.** Repeat the selection of Section 2.6.3 for 100 runs, each with its own 20 training examples. For each run, record the degree chosen by repeated 5-fold cross-validation, by 5-fold cross-validation with a single split, by leave-one-out cross-validation ($K = 20$, for which one split is all there is), and by the test error. How often does each method choose degree 3? When cross-validation chooses a different degree, by what factor does the test error of its choice exceed the test error of degree 3? Explain why a wrong choice is usually cheap here, and describe when it would be expensive.

8. **K-fold cross-validation.**
   1. Suppose that a model trained on m examples has expected generalization error $R(m) = R_\infty + c/m$ with $c > 0$. Each fold's model trains on $(K-1)n/K$ examples. Compute the bias of the K-fold estimate relative to $R(n)$, the error of the model trained on all n examples. Express the bias as a fraction of $R(n) - R_\infty$ and evaluate this fraction for $K = 2$, 5, 10, and n.
   2. After selecting a hyperparameter by cross-validation, the final model is retrained on all n examples. Explain why a hyperparameter whose best value depends on the training-set size complicates this step, and give an example.

9. **Leave-one-out shortcut.** ▯ Leave-one-out cross-validation requires n model fits for a general learner. Consider least squares with the design matrix $\mathbf{X}$, whose rows are $\mathbf{x}^{(i)\top}$. Let $\mathbf{A} = \mathbf{X}^\top\mathbf{X}$ and assume that it remains invertible when any one row of $\mathbf{X}$ is removed. Let $\hat{y}^{(i)}$ be the prediction at $\mathbf{x}^{(i)}$ of the fit on all n examples, let $\hat{y}^{(-i)}$ be the prediction at $\mathbf{x}^{(i)}$ of the fit that omits example i, and let $h_i = \mathbf{x}^{(i)\top}\mathbf{A}^{-1}\mathbf{x}^{(i)}$. Use the Sherman–Morrison identity

$$(\mathbf{A} - \mathbf{u}\mathbf{u}^\top)^{-1} = \mathbf{A}^{-1} + \frac{\mathbf{A}^{-1}\mathbf{u}\mathbf{u}^\top\mathbf{A}^{-1}}{1 - \mathbf{u}^\top\mathbf{A}^{-1}\mathbf{u}} \tag{2.29}$$

with $\mathbf{u} = \mathbf{x}^{(i)}$ to show that

$$y^{(i)} - \hat{y}^{(-i)} = \frac{y^{(i)} - \hat{y}^{(i)}}{1 - h_i}. \tag{2.30}$$

State what leave-one-out cross-validation then costs for least squares compared with a single fit.

2.7 Weight Decay

The polynomial of degree 19 in Section 2.6 passed through its 20 noisy points by using weights of a million or more, and we avoided it by choosing a lower degree on validation data. The degree, however, is a coarse control. Each step adds whole features, and with 20 input variables there are 20 features of degree 1, 210 of degree 2, and 1,540 of degree 3. We would rather keep the features and stop the weights from growing large.

Large weights do damage even in a model with two features. Suppose that the features nearly duplicate each other, such as the same quantity measured by two instruments, and that the label is close to their sum, $y \approx x_1 + x_2$. The weights $(1, 1)$ fit the training data. The weights $(501, -499)$ fit them almost as well: they predict $x_1 + x_2 + 500\,(x_1 - x_2)$, and on the training examples the two measurements differ only slightly. The data say almost nothing about how to split the weight between the two features. Least squares settles the split by fitting the noise in the labels, and since $x_1 - x_2$ is tiny, that takes large weights of opposite sign. On a new example whose two measurements differ by 2, the predictions of the two weight vectors differ by 1,000.

This section adds to the loss a penalty on the size of the weights, whose strength λ we can set on a continuous scale. Among weight vectors that fit the data about equally well, the penalized loss prefers the smallest, $(1, 1)$ in the example. We see what the penalty does to a gradient step, try it on a model with 200 features and 300 noisy examples, and then ask which directions in weight space it shrinks.

2.7.1 Norm Penalties

2.7.1.1 The Penalized Loss

The loss of linear regression in Section 2.1 is the average squared error over the n training examples,

$$L(\mathbf{w}, b) = \frac{1}{n}\sum_{i=1}^{n}\frac{1}{2}\left(\mathbf{w}^\top\mathbf{x}^{(i)} + b - y^{(i)}\right)^2. \tag{2.31}$$

To penalize large weights, we add the squared ℓ_2 norm $\|\mathbf{w}\|^2 = \sum_j w_j^2$ of Section 1.3.4, scaled by a constant $\lambda \geq 0$, and minimize

$$L(\mathbf{w}, b) + \frac{\lambda}{2}\|\mathbf{w}\|^2. \tag{2.32}$$

The first term measures the errors on the training data and the second the size of the weights. The *regularization constant* λ weighs one against the other. With $\lambda = 0$ we recover the original loss. As λ grows, the minimizer moves toward $\mathbf{w} = \mathbf{0}$, the model that ignores its inputs and predicts b everywhere. Note that the training loss cannot choose λ for us, since it is always lowest at $\lambda = 0$. Like the degree of a polynomial in Section 2.6.3, λ is a hyperparameter, chosen on validation data.

The factor $\frac{1}{2}$ is merely convenient: it makes the gradient of the penalty $\lambda\mathbf{w}$. Leaving out the bias, by contrast, is deliberate. Adding a constant to every label should move the fit by that constant at no cost, whereas a penalty on b would pull every prediction toward zero rather than toward the average label.

A change to training whose purpose is to lower the error on new data rather than on the training data is called *regularization*. The penalty Equation 2.32 is the most common regularizer in deep learning, where it is called weight decay (Hanson and Pratt 1988; Krogh and Hertz 1992) for a reason we see shortly. Statisticians call linear regression with this penalty *ridge regression* (Hoerl and Kennard 1970).

What does a smaller norm change about the model? For a linear model, the norm bounds how strongly the prediction can respond to a change of the input: by the Cauchy–Schwarz inequality $|\mathbf{u}^\top\mathbf{v}| \leq \|\mathbf{u}\|\,\|\mathbf{v}\|$ of Section 1.3, the predictions for two inputs $\mathbf{x}$ and $\mathbf{x}'$ differ by at most

$$|\mathbf{w}^\top\mathbf{x} - \mathbf{w}^\top\mathbf{x}'| \leq \|\mathbf{w}\|\,\|\mathbf{x} - \mathbf{x}'\|.$$

A model with small weights cannot amplify a small change of its input, such as noise in a measurement. In the opening example, the weights $(1, 1)$ have norm $\sqrt{2} \approx 1.4$ and the weights $(501, -499)$ norm about 707, and only the second pair can turn a difference of 2 between the measurements into a change of 1,000 in the prediction. This reading relies on the model being linear, so that its weights are the slopes of the function it computes; in the networks of later chapters, the same function can be computed by parameters of very different norms.

2.7.1.2 Weight Decay in the Gradient Step

What does the penalty do to training? Adding its gradient $\lambda\mathbf{w}$ to the minibatch SGD update of Section 2.1.2.3, with learning rate η and minibatch $\mathcal{B}_t$, gives

$$\mathbf{w} \leftarrow (1 - \eta\lambda)\,\mathbf{w} - \frac{\eta}{|\mathcal{B}|}\sum_{i \in \mathcal{B}_t} \mathbf{x}^{(i)}\left(\mathbf{w}^\top\mathbf{x}^{(i)} + b - y^{(i)}\right), \tag{2.33}$$

while the update of b does not change. The update first multiplies the weights by $1 - \eta\lambda$, a number slightly below one, and then takes the usual step toward fitting the minibatch. Without the data term, the weights would decay geometrically, to $(1 - \eta\lambda)^t$ times their

initial value after t steps, which is why the method is called *weight decay*. With $\eta = 0.05$ and $\lambda = \frac{1}{3}$, the values of our experiment, each step keeps $1 - \frac{1}{60} \approx 98.3\%$ of the weights, and an epoch of 15 steps keeps $0.983^{15} \approx 78\%$. The data term pulls back wherever the training data favor nonzero weights, strongly in some directions and weakly in others, and the weights settle where the two pulls balance.

For minibatch SGD, adding $\frac{\lambda}{2}\|\mathbf{w}\|^2$ to the loss and multiplying the weights by $1-\eta\lambda$ in every step are the same operation, and frameworks apply the decay inside the optimizer rather than through the loss. The two forms part ways for optimizers that rescale each coordinate of the gradient (exercise 2). Shrinking the weights directly, whatever the optimizer does with the gradient, is called *decoupled weight decay* (Loshchilov and Hutter 2019), and Section 9.5.4 shows why the distinction matters for adaptive optimizers.

2.7.1.3 The ℓ_1 Penalty and Sparsity

Other norms give other penalties. With the ℓ_1 norm $\|\mathbf{w}\|_1 = \sum_j |w_j|$ in place of the squared ℓ_2 norm, linear regression becomes *lasso* regression (Tibshirani 1996). When the two duplicated measurements of the opening agree, the weights $(1, 1)$ and $(2, 0)$ make the same predictions. The squared ℓ_2 norm prefers the even split, 2 against 4, while the ℓ_1 norm is 2 for both and for every split in between.

The difference is easiest to see with the penalty written as a constraint. Minimizing the penalized loss is equivalent to minimizing L subject to $\|\mathbf{w}\| \le s$ for ridge, or $\|\mathbf{w}\|_1 \le s$ for lasso, with a bound s that depends on λ (Section 35.7.3.2). The constrained solution $\hat{\mathbf{w}}_\lambda$ is the point where the smallest contour of the loss around the unconstrained minimizer $\hat{\mathbf{w}}$ touches the constraint region (Figure 2.10). The ℓ_2 ball is round, so the contact point usually has every coordinate nonzero: ridge regression shrinks the weights but rarely sets one to zero. The ℓ_1 ball is a diamond with its corners on the axes, and a contour often touches it first at a corner, where some coordinates are zero. Lasso therefore selects features, which helps when only a few of many features matter. The rest of this section uses the ℓ_2 penalty, whose minimizer has a closed form.

2.7.1.4 A Gaussian Prior on the Weights

The penalty also has a probabilistic reading, and with it comes an opinion on how large λ should be. In Section 2.1.3 we derived the squared loss from Gaussian noise. If each label is $y = \mathbf{w}^\top \mathbf{x} + b + \epsilon$ with $\epsilon \sim \mathcal{N}(0, \sigma^2)$, the negative log-likelihood of the training labels is

$$-\log p(\mathbf{y} \mid \mathbf{X}, \mathbf{w}, b) = \frac{1}{2\sigma^2} \sum_{i=1}^{n} \left(y^{(i)} - \mathbf{w}^\top \mathbf{x}^{(i)} - b\right)^2 + \text{const},$$

and maximum likelihood minimizes it.

Maximum likelihood uses only the data. Suppose that before seeing any data we believe large weights to be unlikely. We can state this belief as a *prior* distribution on the weights, here independent normal weights $w_j \sim \mathcal{N}(0, \tau^2)$, whose standard deviation τ says how large we expect them to be.

Bayes' theorem (Section 1.6) combines the prior with the likelihood into the *posterior* $p(\mathbf{w} \mid \mathbf{X}, \mathbf{y}) \propto p(\mathbf{y} \mid \mathbf{X}, \mathbf{w})\, p(\mathbf{w})$, the distribution of the weights after seeing the data. The weights

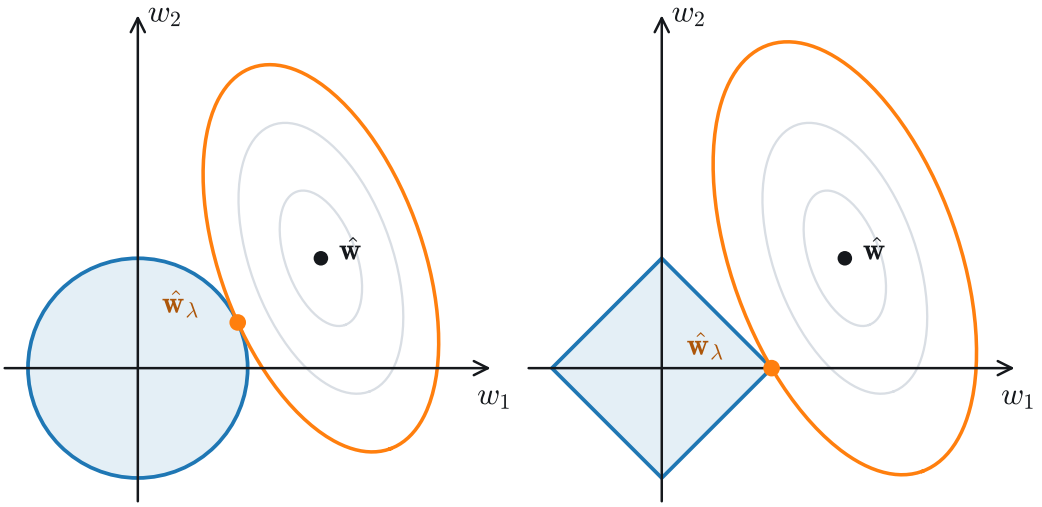


Figure 2.10: Weight decay as a constraint. The contours of the training loss are ellipses around the unconstrained minimizer $\hat{\mathbf{w}}$, and the regularized solution $\hat{\mathbf{w}}_\lambda$ is the point where the smallest contour (orange) touches the constraint region. Left: the ℓ_2 ball is touched where both coordinates are nonzero. Right: the ℓ_1 diamond is touched at a corner, where $w_2 = 0$.

with the highest posterior density are the *maximum a posteriori* (MAP) estimate. In the negative logarithm of the posterior, the prior contributes

$$-\log p(\mathbf{w}) = \frac{1}{2\tau^2}\|\mathbf{w}\|^2 + \text{const}, \tag{2.34}$$

a quadratic penalty on the weights. Multiplying the negative log-posterior by σ^2/n, which does not move its minimizer, turns it into the penalized loss Equation 2.32 with

$$\lambda = \frac{\sigma^2}{n\tau^2}. \tag{2.35}$$

Weight decay is thus MAP estimation under a Gaussian prior on the weights, and the formula says how strong the penalty should be. It grows with the noise level σ, since noisy labels deserve less trust, and it grows as the prior narrows, with smaller τ. It falls as $1/n$, since more data outweigh the prior. The appendix treats MAP estimation in general (Section 36.6.4.1).

2.7.2 Weight Decay from Scratch

Let us try the penalty on a problem with many weights and noisy labels. The data come from a linear model with $d = 200$ standard normal features,

$$y = 0.05 + \sum_{j=1}^{200} 0.1\, x_j + \epsilon, \quad \epsilon \sim \mathcal{N}(0, 1), \tag{2.36}$$

so the true weights are $\mathbf{w}^* = (0.1, \ldots, 0.1)$ and the true bias is $b^* = 0.05$. The weights have the size $\tau = 0.1$ that the prior describes. The noise is large: its standard deviation $\sigma = 1$ is not much smaller than that of the noise-free label, $\|\mathbf{w}^*\| = \sqrt{2} \approx 1.4$. We train on 300 examples, in minibatches of 20, and validate on 1,000. Three hundred examples are enough to determine all 200 weights, but, as we will see, some combinations of them only poorly. `SyntheticRegressionData` from Section 2.4 draws such data.

```
tau, sigma, n = 0.1, 1.0, 300   # size of the weights, noise, examples
data = d2l.SyntheticRegressionData(w=d2l.ones(200) * tau, b=0.05,
                                   noise=sigma, num_train=n, num_val=1000,
                                   batch_size=20)
```

What validation loss should a useful model reach? By Equation 2.22, the expected loss is half the squared distance of the parameters from the true ones, plus $\sigma^2/2$. A model that ignores its inputs and predicts b^* everywhere therefore has loss $\frac{1}{2}(\|\mathbf{w}^*\|^2 + \sigma^2) = \frac{1}{2}(2 + 1) = 1.5$, and the true parameters have loss $\sigma^2/2 = 0.5$. A model is worth its 200 weights only if it lands clearly below 1.5.

2.7.2.1 The Penalized Model

The penalty is half the sum of the squared weights.

```
def l2_penalty(w):
    return d2l.reduce_sum(w**2) / 2
```

We extend `LinearRegressionScratch`, the model of Section 2.3 written as a `Module` in Section 2.4. Only `training_step` changes: it returns the penalized loss Equation 2.32, which the trainer differentiates and minimizes. `validation_step` still returns the plain squared loss, so that validation losses at different values of λ all measure the quality of the predictions.

```
class WeightDecayScratch(d2l.LinearRegressionScratch):
    def __init__(self, num_inputs, lambd, lr, sigma=0.01):
        super().__init__(num_inputs, lr, sigma)
        self.save_hyperparameters()

    def training_step(self, batch):
        loss = super().training_step(batch)
        return loss + self.lambd * l2_penalty(self.w)
```

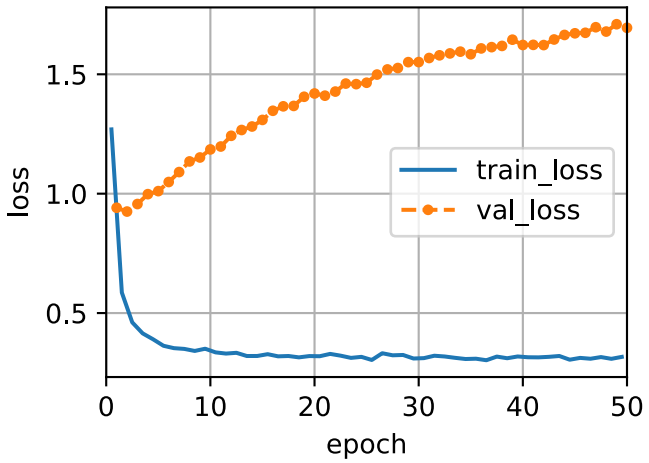
2.7.2.2 Training with and without the Penalty

`train_scratch` trains a model with regularization constant `lambd` (`lambda` is a reserved word in Python) on `data` for 50 epochs at learning rate 0.05, which makes 750 SGD steps. It returns the model, its validation loss after each epoch, read from `trainer.history`, and the norm of the learned weights. The plot shows the training and validation losses of each epoch.

```
def train_scratch(lambd, data):
    trainer = d2l.Trainer(max_epochs=50)
    model = WeightDecayScratch(num_inputs=200, lambd=lambd, lr=0.05)
    trainer.fit(model, data)
    norm = float(torch.linalg.vector_norm(model.w.detach()))
    return model, [float(v) for v in trainer.history['val_loss']], norm
```

We start without the penalty.

```
model_0, val_losses, norm = train_scratch(0, data)
```



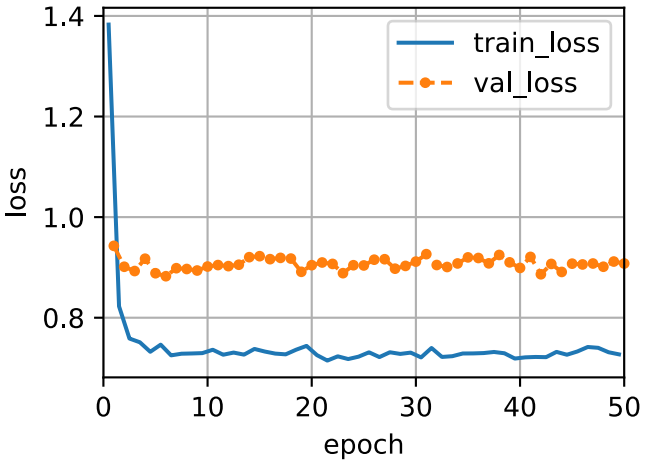

```
print(f'validation loss {val_losses[-1]:.3f} (lowest {min(val_losses):.3f}, '
      f'epoch {val_losses.index(min(val_losses)) + 1}), norm of w {norm:.2f}')
```

```
validation loss 1.695 (lowest 0.925, epoch 2), norm of w 2.06
```

The training loss falls to about 0.2 to 0.3, well below the 0.5 of the true parameters: the model fits part of the noise in its 300 training labels. The validation loss is lowest after two or three epochs, at about 0.9, and then climbs, usually to between about 1.2 and 1.9; runs that end lower are rare. After 50 epochs the model predicts about as well as the constant b^*, which scores 1.5, and in some runs worse. A model with 200 weights that cannot beat a constant has learned mostly noise. Why its loss is lowest so early, after only 30 or 45 steps, is a question we can answer once we know which directions in weight space training fits first; we return to it at the end of the section.

Now we add the penalty. Which λ? The Gaussian prior answers: with $\sigma = 1$, $n = 300$, and weights of size $\tau = 0.1$, it asks for $\lambda = \sigma^2/(n\tau^2) = \frac{1}{3}$.

```
lambd = sigma**2 / (n * tau**2)   # the value the prior suggests, 1/3
model_lambd, val_losses, norm = train_scratch(lambd, data)
```




```
print(f'validation loss {val_losses[-1]:.3f}, norm of w {norm:.2f}')
```

```
validation loss 0.908, norm of w 1.25
```

The training curve now shows the penalized loss, which stays above 0.5: the penalized model no longer chases the noise in the training labels. The validation loss settles between about 0.8 and 1.0 and stays there, well below the 1.5 of the constant prediction. The norm of the weights shrinks from about 2 to about 1.2, below the norm 1.4 of $\mathbf{w}^*$. The penalized weights are shorter than the true ones, and still they predict far better than the longer unpenalized weights.

We should be candid about what was gained. Measured against the unpenalized model after 50 epochs, the penalty removes a quarter to a half of the validation loss. Measured against the same model stopped at its best epoch, it removes almost nothing: both land near 0.9. What the penalty changes is the loss being minimized, so its result holds for as long as we care to train, whereas the unpenalized run has to be stopped at a moment that its training loss does not announce. Which part of the weights did the penalty change?

2.7.3 Shrinkage Along Eigenvectors

For linear regression we can say exactly what the penalty removed, because the minimizer of the penalized loss has a closed form. It leaves the directions in weight space that the data determine well nearly untouched. Along the directions that the data determine poorly, it damps the large coordinates that least squares picks up from noise in the labels. We derive this first, then watch it on the duplicated features, which have one poorly determined direction, and finally return to the experiment.

2.7.3.1 The Ridge Estimator

To keep the formulas short, we drop the bias in this section. With an unpenalized bias, the same analysis applies after the features and the labels are centered (exercise 2). Without the bias, the loss is $L(\mathbf{w}) = \frac{1}{2n}\|\mathbf{X}\mathbf{w} - \mathbf{y}\|^2$, with gradient $\frac{1}{n}\mathbf{X}^\top(\mathbf{X}\mathbf{w} - \mathbf{y})$. Adding the gradient $\lambda\mathbf{w}$ of the penalty and setting the sum to zero gives

$$\left(\frac{1}{n}\mathbf{X}^\top\mathbf{X} + \lambda\mathbf{I}\right)\mathbf{w} = \frac{1}{n}\mathbf{X}^\top\mathbf{y}. \tag{2.37}$$

At $\lambda = 0$ these are the normal equations of Section 2.1.2.1, divided by n. They have many solutions when the columns of $\mathbf{X}$ are dependent, as they are whenever there are fewer examples than features. For $\lambda > 0$ the matrix on the left is invertible: for every $\mathbf{v} \neq \mathbf{0}$, $\mathbf{v}^\top\left(\frac{1}{n}\mathbf{X}^\top\mathbf{X} + \lambda\mathbf{I}\right)\mathbf{v} = \frac{1}{n}\|\mathbf{X}\mathbf{v}\|^2 + \lambda\|\mathbf{v}\|^2 > 0$, so the matrix maps no nonzero vector to zero. The penalized loss therefore has exactly one minimizer, the *ridge estimator* $\hat{\mathbf{w}}_\lambda$.

2.7.3.2 Shrinkage Factors

The matrix $\frac{1}{n}\mathbf{X}^\top\mathbf{X}$ is symmetric, so it has perpendicular eigenvectors $\mathbf{v}_1, \dots, \mathbf{v}_d$ of length 1, one per dimension, with real eigenvalues $\mu_1, \dots, \mu_d$ (Section 1.3.5). Each eigenvalue measures how strongly the training data determine the weights along its eigenvector. Since $\mu_j = \mathbf{v}_j^\top\left(\frac{1}{n}\mathbf{X}^\top\mathbf{X}\right)\mathbf{v}_j = \frac{1}{n}\|\mathbf{X}\mathbf{v}_j\|^2$, moving the weights a unit distance along $\mathbf{v}_j$ changes the predictions on the training inputs by μ_j in mean square. The eigenvalues are never negative, and $\mu_j = 0$ means that the move changes no training prediction at all.

In the basis of eigenvectors, the equations Equation 2.37 separate. We write the weights as $\mathbf{w} = \sum_j c_j\mathbf{v}_j$ and the right-hand side as $\frac{1}{n}\mathbf{X}^\top\mathbf{y} = \sum_j g_j\mathbf{v}_j$. The matrix maps $\mathbf{v}_j$ to $(\mu_j+\lambda)\,\mathbf{v}_j$, so the equations become $(\mu_j+\lambda)\,c_j = g_j$, one for each j. Without the penalty, least squares gives $c_j = g_j/\mu_j$ wherever $\mu_j > 0$. With the penalty,

$$c_j = \frac{\mu_j}{\mu_j + \lambda}\cdot\frac{g_j}{\mu_j}.$$

The ridge estimator multiplies the least-squares coordinate along each eigenvector by the *shrinkage factor* $\mu_j/(\mu_j + \lambda)$.

The factor is close to 1 when $\mu_j \gg \lambda$, close to 0 when $\mu_j \ll \lambda$, and $\frac{1}{2}$ at $\mu_j = \lambda$ (Figure 2.11). Directions that the data determine well keep their coordinates, and directions that they determine poorly are suppressed. In the poorly determined directions least squares is easily misled: noise in the labels changes g_j a little, and the least-squares coordinate g_j/μ_j a lot when μ_j is small.

Directions with $\mu_j = 0$ are the extreme case. There $g_j = \frac{1}{n}\mathbf{v}_j^\top\mathbf{X}^\top\mathbf{y} = \frac{1}{n}(\mathbf{X}\mathbf{v}_j)^\top\mathbf{y} = 0$, so the least-squares equation $\mu_j c_j = g_j$ reads $0 = 0$ and leaves c_j undetermined, while the ridge estimator sets $c_j = 0$ for every $\lambda > 0$. Every model with more weights than examples has such directions (exercise 8).

Adding up the shrinkage factors gives the *effective number of parameters* (statisticians say degrees of freedom),

$$\mathrm{df}(\lambda) = \sum_{j=1}^{d}\frac{\mu_j}{\mu_j + \lambda}. \tag{2.38}$$

A direction counts as one parameter if it is kept, as none if it is suppressed, and as a fraction in between. As λ decreases to 0, df approaches the number of nonzero eigenvalues, and as λ grows it falls continuously to 0. This is the continuous control of capacity we asked for at

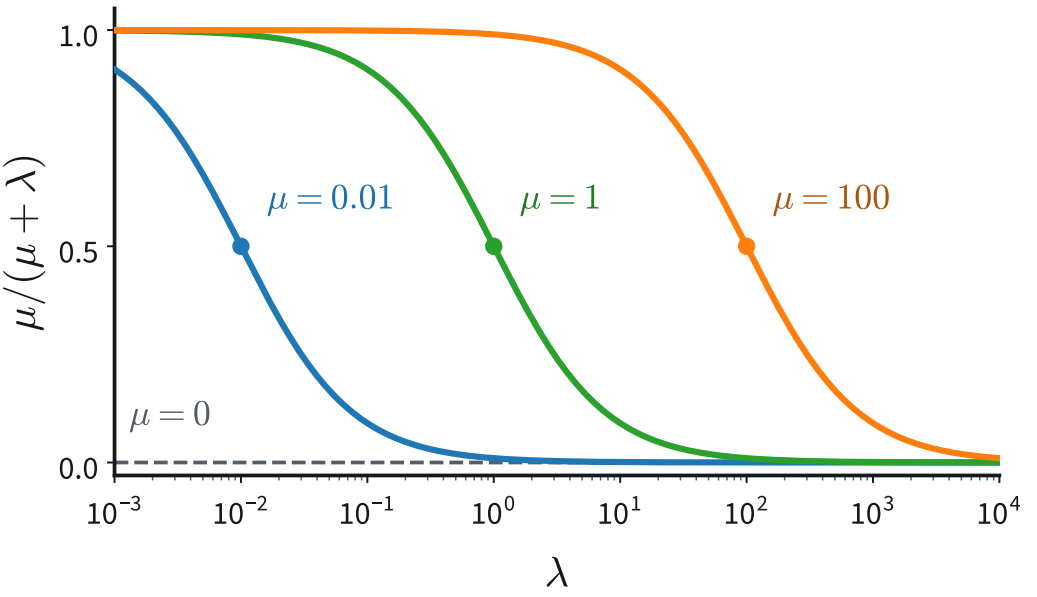


Figure 2.11: The shrinkage factor $\mu/(\mu + \lambda)$ of the ridge estimator along an eigenvector with eigenvalue μ, against the regularization constant λ. A direction keeps its coordinate while $\lambda \ll \mu$ and loses it once $\lambda \gg \mu$; the factor is $\frac{1}{2}$ at $\lambda = \mu$ (dots). Directions with $\mu = 0$ are set to zero for every $\lambda > 0$ (dashed).

the start of the section: instead of adding or removing whole features, λ moves the effective number of parameters through every value in between.

2.7.3.3 Two Duplicated Features

Let us return to the duplicated features of the opening and watch the damping with numbers. We draw $n = 100$ examples with $x_1 \sim \mathcal{N}(0, 1)$ and a second feature $x_2 = x_1 + 0.001\, z$ with $z \sim \mathcal{N}(0, 1)$, label them $y = x_1 + x_2 + \epsilon$ with $\epsilon \sim \mathcal{N}(0, 0.1^2)$, and fit without a bias. The noise, $\sigma = 0.1$, is ten times smaller than in our experiment, so here the labels are almost exact. The sum direction $(1, 1)/\sqrt{2}$ then has an eigenvalue between about 1 and 3, and the difference direction $(1, -1)/\sqrt{2}$ one below 10^{-6}, so at $\lambda = 0.01$ ridge keeps the sum and removes the difference. NumPy suffices here: it draws the data and solves both problems in closed form. We test each fit on 1,000 new examples drawn in the same way and on 1,000 whose features disagree more, $x_2 = x_1 + 0.1\, z$.

```
rng = np.random.default_rng()  # A new draw on every run

def twins(n, gap):
    """Two near-copies of a feature, x2 = x1 + gap * noise, labeled by
    y = x1 + x2 + noise with standard deviation 0.1."""
    x1 = rng.normal(size=n)
    X = np.stack([x1, x1 + gap * rng.normal(size=n)], axis=1)
    return X, X.sum(axis=1) + 0.1 * rng.normal(size=n)
```

```
def mse(w, data):
    """Mean squared error of the weights w on data = (X, y)."""
    X, y = data
    return ((X @ w - y) ** 2).mean()

def ridge(X, y, lambd):
    """The ridge estimator: the solution of eq_wd-ridge-normal."""
    n, d = X.shape
    return np.linalg.solve(X.T @ X / n + lambd * np.eye(d), X.T @ y / n)
```

`ridge` solves the equations Equation 2.37. We fit two training sets, each by least squares and by ridge regression.

```
near, far = twins(1000, 0.001), twins(1000, 0.1)  # the two test sets
for _ in range(2):                                # two training sets
    X, y = twins(100, 0.001)
    w_ls = np.linalg.lstsq(X, y, rcond=None)[0]
    w_ridge = ridge(X, y, lambd=0.01)
    for name, w in (('least squares', w_ls), ('ridge', w_ridge)):
        print(f'{name:>13} ({w[0]:6.2f}, {w[1]:6.2f}): '
              f'test {mse(w, near):.3f}, disagreeing {mse(w, far):.3f}')
mu = np.linalg.eigvalsh(X.T @ X / len(X))         # in increasing order
print(f'eigenvalues of X^T X / n: {mu[0]:.1e}, {mu[1]:.1e}')
```

```
least squares (  5.59,  -3.59): test 0.010, disagreeing 0.213
        ridge (  0.99,   0.99): test 0.010, disagreeing 0.010
least squares ( 13.88, -11.87): test 0.010, disagreeing 1.613
        ridge (  1.00,   1.00): test 0.010, disagreeing 0.010
eigenvalues of X^T X / n: 6.0e-07, 2.0e+00
```

Least squares returns different weights for each training set. They always add up to about 2, the sum that the data determine, but the split between them changes from draw to draw, often by several units and sometimes by tens, with one weight then large and the other negative. Even almost exact labels hold enough noise to push the weights that far. Ridge returns about $(1, 1)$ every time. On new examples like the training examples the difference is invisible: both fits have a test error of about 0.01, the noise variance 0.1^2 of this example. It shows on examples whose features disagree more, where least squares fails whenever its weights lie far from 1. Since $w_2 - 1 \approx -(w_1 - 1)$, its error on such an example is about $(w_1 - 1)(x_1 - x_2)$ plus the noise, with mean square about $0.01\,(w_1 - 1)^2 + 0.01$. The large weights amplify the disagreement, just as the Cauchy–Schwarz bound allows. Ridge keeps its error at about 0.01. This is the situation the penalty is made for: a direction that the data barely determine, and noise ready to fill it.

2.7.3.4 The Directions Our Data Determine

Was the experiment with 200 features such a situation? To find out, we compute the eigenvalues of $\frac{1}{n}\mathbf{X}^\top\mathbf{X}$ for its 300 training inputs (uncentered, which changes the numbers only slightly), and the shrinkage factors at $\lambda = \frac{1}{3}$. As in Section 1.3, `eigh` returns the eigenvalues in increasing order and the eigenvectors as the columns of its second result. We count the eigenvalues below λ, the directions in which the penalty removes more than half of the

least-squares coordinate.

```
X = data.X[:n]                                # the 300 training inputs
mu, V = torch.linalg.eigh(X.T @ X / n)        # increasing eigenvalues
factors = mu / (mu + lambd)                   # shrinkage at lambda = 1/3
k = int((mu < lambd).sum())                   # the poorly determined ones
print(f'mu from {float(mu[0]):.2f} to {float(mu[-1]):.1f}, '
      f'{k} of {len(mu)} below lambda')
print(f'factors from {float(factors[0]):.2f} to {float(factors[-1]):.2f}, '
      f'df = {float(factors.sum()):.0f}')
```

```
mu from 0.04 to 3.1, 54 of 200 below lambda
factors from 0.11 to 0.90, df = 127
```

No eigenvalue is zero: 300 examples determine every direction of the 200-dimensional weight space. They determine some far better than others, though. The eigenvalues run from a few hundredths to about 3.2, and roughly a quarter of them, 50 to 60, lie below λ. The shrinkage factors run from about 0.1 to 0.9, and $\mathrm{df}(\frac{1}{3}) \approx 127$: the regularized model has 200 weights but about 127 effective parameters.

How much does a small eigenvalue hurt least squares? The noise in the labels contributes $\frac{1}{n}(\mathbf{X}\mathbf{v}_j)^\top \epsilon$ to g_j, an amount with standard deviation $\sigma\sqrt{\mu_j/n}$, since $\|\mathbf{X}\mathbf{v}_j\|^2 = n\mu_j$. Divided by μ_j, it moves the least-squares coordinate g_j/μ_j by about $\sigma/\sqrt{n\mu_j}$: by 0.06 when $\mu_j = 1$, but by 0.3 when $\mu_j = 0.04$. The true coordinates are about $\|\mathbf{w}^*\|/\sqrt{200} = 0.1$ in size, so along the directions with the smallest eigenvalues least squares returns mostly noise. The validation data, whose features are standard normal, weigh every direction alike: by Equation 2.22, an error c_j along $\mathbf{v}_j$ raises the expected loss by $c_j^2/2$, whatever μ_j is. We therefore split the error $\mathbf{w} - \mathbf{w}^*$ of both trained models into its coordinates along the eigenvectors, `V.T @ (w - w_star)`, and add up their squares over the k poorly determined directions and over the rest.

```
w_star = d2l.reshape(data.w, (-1, 1))
for name, model in (('lambda = 0', model_0), ('lambda = 1/3', model_lambd)):
    c = V.T @ (model.w.detach() - w_star)     # error along each eigenvector
    print(f'{name:>12}: squared error {float((c[:k] ** 2).sum()):.2f} in the '
          f'{k} poorly determined directions, '
          f'{float((c[k:] ** 2).sum()):.2f} in the rest')
```

```
  lambda = 0: squared error 1.81 in the 54 poorly determined directions, 0.68 in the rest
lambda = 1/3: squared error 0.42 in the 54 poorly determined directions, 0.46 in the rest
```

Without the penalty, roughly a quarter of the directions hold more than half of the squared error, between about 0.9 and 2.2, against about 0.5 in all the others. The penalty cuts that part by a factor of two to six and the rest a little, leaving about 0.3 to 0.5 in each. Even along the better determined directions the noise is not small next to the true coordinates, and a factor between 0.5 and 0.9 removes more noise than signal. As a check, half the total squared error plus $\sigma^2/2 = 0.5$ comes close to the validation loss that we measured for each model. Weight decay did here, in some fifty directions at once, what it did for the difference of the two duplicated features.

The eigenvalues also answer the question we left open: why the unpenalized validation loss

was lowest after two or three epochs. A gradient step on the unpenalized loss multiplies the gap between the coordinate along $\mathbf{v}_j$ and its least-squares value by $1 - \eta\mu_j$. After one epoch of 15 steps at $\eta = 0.05$, a direction with $\mu_j = 3.2$ has closed $1 - 0.84^{15} \approx 93\%$ of its gap, and a direction with $\mu_j = 0.04$ only $1 - 0.998^{15} \approx 3\%$. After two epochs the model holds the well-determined coordinates and almost none of the noisy ones, much as the ridge estimator does; as training goes on, the poorly determined coordinates fill up with their noise and the validation loss climbs. Stopping early is therefore a regularizer too, and in this run as good a one as the penalty. It is harder to control, since the right moment depends on the learning rate and on the minibatch noise and must be read off validation data in every run (exercise 6).

2.7.4 Concise Implementation

Weight decay is so common that deep learning frameworks build the update Equation 2.33 into their optimizers, so no penalty needs to appear in the loss. They also let different parameters follow different rules, which is how we decay the weights but not the bias.

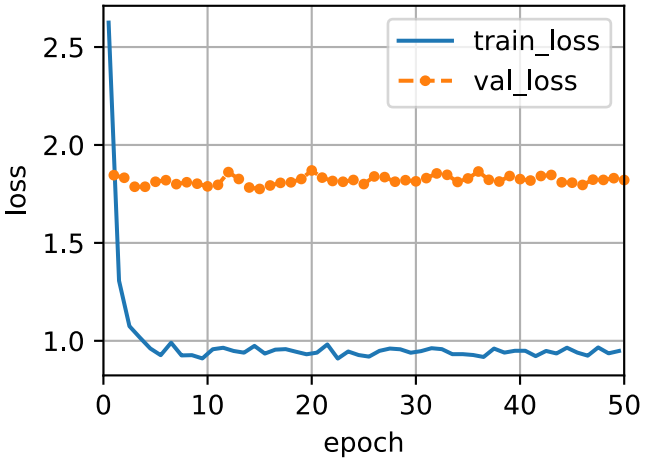
PyTorch optimizers take a `weight_decay` argument. Passing the parameters in two groups decays the weight `net.weight` but not the bias `net.bias`. As in Section 2.5, we initialize the weights from $\mathcal{N}(0, 0.01^2)$ and the bias with zero, by applying `weight_init` to the layer once its first call has created the parameters.

```
class WeightDecay(d2l.LinearRegression):
    def __init__(self, wd, lr):
        super().__init__(lr)
        self.save_hyperparameters()

    def configure_optimizers(self):
        return torch.optim.SGD([
            {'params': self.net.weight, 'weight_decay': self.wd},
            {'params': self.net.bias}], lr=self.lr)
```

To reproduce `train_scratch(lambd, data)` we have to convert both hyperparameters. The framework loss lacks our factor $\frac{1}{2}$, so we halve the learning rate to 0.025, as in Section 2.5. The decay term $\lambda\mathbf{w}$, however, comes from the optimizer and is not doubled along with the gradient of the loss. At half the learning rate the weight decay must therefore double, to $2\lambda = \frac{2}{3}$, for each step to keep the factor $1 - 0.025 \cdot \frac{2}{3} = 1 - \frac{1}{60}$.

```
trainer = d2l.Trainer(max_epochs=50)
model = WeightDecay(wd=2 * lambd, lr=0.025)
model(data.X[:1])  # Creates the parameters of the lazy layer
model.net.apply(model.weight_init)
trainer.fit(model, data)
norm = float(torch.linalg.vector_norm(model.get_w_b()[0]))
```




```
print(f'validation loss {trainer.history["val_loss"][-1]:.3f}, '
      f'norm of w {norm:.2f}')
```

```
validation loss 1.821, norm of w 1.25
```

The norm of the weights, about 1.2, matches the scratch run at $\lambda = \frac{1}{3}$, and so does the validation loss: between about 1.6 and 1.9, it is the framework's mean squared error, twice our loss. The training curve shows the plain mean squared error rather than the penalized loss, because the decay lives in the optimizer, not in the loss.

In practice, weight decay is this one optimizer argument. Outside a synthetic experiment we rarely know σ and τ, so λ is chosen on validation data, one training run per candidate. What the argument controls in a deep network, whose parameters are no longer the slopes of the function, is a harder question, and the bound for linear models does not answer it.

2.7.5 Summary

Weight decay adds the penalty $\frac{\lambda}{2}\|\mathbf{w}\|^2$ to the loss, so that among weight vectors that fit the data about equally well, training prefers the smallest. Under minibatch SGD the penalty multiplies the weights by $1-\eta\lambda$ in every step, and frameworks apply it inside the optimizer. The penalty is MAP estimation under a Gaussian prior, which suggests $\lambda = \sigma^2/(n\tau^2)$, and it shrinks weights where the ℓ_1 penalty of lasso sets them to zero. For linear regression it multiplies the coordinate along each eigenvector of $\frac{1}{n}\mathbf{X}^\top\mathbf{X}$ by $\mu_j/(\mu_j+\lambda)$, so that directions the data determine well survive, poorly determined ones are damped, and λ moves the effective number of parameters continuously. For two duplicated features, ridge returned weights near $(1, 1)$ where least squares returned large weights of opposite sign. With 200 features and 300 noisy examples, the prior's λ cut the validation loss by a quarter to a half, mostly by damping the quarter of the directions that the data determine poorly. Stopping the unpenalized run after two epochs does nearly as well, because gradient descent fits those directions last, but the moment has to be found on validation data.

2.7.6 Exercises

1. **The ℓ_1 update.** Replace the penalty $\frac{\lambda}{2}\|\mathbf{w}\|^2$ by $\lambda\sum_j |w_j|$.

   1. Derive the minibatch SGD update for $\mathbf{w}$, using the subgradient $\operatorname{sign}(w_j)$ of exercise 3 in place of the derivative of $|w_j|$.
   2. Compare the change that the penalty alone makes to w_j in one step of this update with the change it makes in the update Equation 2.33, once for large and once for small $|w_j|$.
   3. The update of (a) almost never sets w_j exactly to zero. Split it into a gradient step on L followed by the penalty step, and modify the penalty step so that it sets w_j to zero whenever it would change the sign of w_j. Relate the result to the feature selection that lasso performs. The modified step is *soft-thresholding*, the proximal step of the ℓ_1 norm (Section 35.6.4.1).

2. **Shrinkage versus penalty.** Weight decay can be applied by shrinking the weights before the gradient step or by adding $\frac{\lambda}{2}\|\mathbf{w}\|^2$ to the loss L.

   1. Show that the update $\mathbf{w} \leftarrow (1-\eta\lambda)\mathbf{w} - \eta\nabla L$ is identical to a gradient step on the penalized loss $L + \frac{\lambda}{2}\|\mathbf{w}\|^2$.
   2. Suppose that the gradient is rescaled coordinate-wise by fixed constants $a_j > 0$ before the step. Shrinking and then stepping gives the first update below, and stepping on the penalized gradient gives the second:

$$w_j \leftarrow (1-\eta\lambda)\, w_j - \eta a_j\, \partial_{w_j} L \qquad \text{and} \qquad w_j \leftarrow w_j - \eta a_j \left(\partial_{w_j} L + \lambda w_j\right). \tag{2.39}$$

      Determine for which a_j the two coincide, and the factor by which the second update shrinks w_j in a step with $\partial_{w_j} L = 0$. Many optimizers used in deep learning rescale the gradient in this way. Which of the two updates keeps λ a rate of decay that is the same for every weight?

   Adapted from Simon Prince, Understanding Deep Learning, Problem 9.5.

3. **MAP estimation.** For the prior $\mathbf{w} \sim \mathcal{N}(\mathbf{0}, \tau^2\mathbf{I})$ and noise variance σ^2, this section derived $\lambda = \sigma^2/(n\tau^2)$.

   1. The duplicated features of Section 2.7.3 have noise $\sigma = 0.1$, $n = 100$ examples, and true weights of size
      1. Compute the λ that the prior suggests, and compare it with the value $\lambda = 0.01$ used there. Use the two eigenvalues of that example to explain why both values return weights near $(1, 1)$.
   2. Holding τ and σ fixed, describe how λ changes as n grows, and interpret the change in terms of the relative weight of prior and likelihood.
   3. The bias b is not penalized. Which prior on b does this correspond to?

4. code **Ridge as augmented least squares.** Center the 300 training inputs and labels so that the bias drops out, and compute the ridge estimator for $\lambda = \frac{1}{3}$ in two ways.

   1. Solve the linear system Equation 2.37 with `ridge`.
   2. Run ordinary least squares with `lstsq` on $\mathbf{X}$ stacked on top of $\sqrt{n\lambda}\,\mathbf{I}$, with the labels $\mathbf{y}$ followed by d zeros.

   Confirm that the two agree up to rounding errors, and explain why. Then compare the norm of the result with the norm that `train_scratch(lambd, data)` returned, and name the differences between the two procedures that could account for any gap.

Adapted from Hastie, Tibshirani, and Friedman, The Elements of Statistical Learning, Exercise 3.12.

5. code **The λ sweep.** Train `WeightDecayScratch` on `data` for $\lambda \in \{0.01, 0.1, 0.3, 1, 3, 10, 30\}$.
   1. Plot the final validation loss and the norm of $\mathbf{w}$ against λ on a logarithmic axis, with the values of `train_scratch(0, data)` as horizontal lines, and report the λ with the smallest validation loss.
   2. Compute $\mathrm{df}(\lambda)$ for the same values from the eigenvalues of Section 2.7.3, and plot the validation loss against $\mathrm{df}(\lambda)$. Compare the shape with Figure 2.8.
   3. Is the value $\lambda = \frac{1}{3}$ that the prior suggests close to the best value of the sweep? Generate data with noise $\sigma = 0.5$ and with $\sigma = 2$, recompute the prior's λ for each, and check whether it follows the best value of a sweep.

6. code **Stopping early versus decay.** The unpenalized run of this section reaches its lowest validation loss after two or three epochs, about as low as the run with $\lambda = \frac{1}{3}$ ends.
   1. From weights near zero, gradient descent on the unpenalized loss multiplies the gap between the coordinate along $\mathbf{v}_j$ and its least-squares value by $1 - \eta\mu_j$ per step, so after t steps it has built up the fraction $1 - (1 - \eta\mu_j)^t$ of that coordinate. Plot this factor against μ_j for the eigenvalues of Section 2.7.3, with t equal to one, two, and fifty epochs of 15 steps at $\eta = 0.05$, next to the ridge factor $\mu_j/(\mu_j + \lambda)$ at $\lambda = \frac{1}{3}$. Which t comes closest to ridge?
   2. Train `WeightDecayScratch` with $\lambda = 0$ for two epochs and split its error along the eigenvectors as Section 2.7.3 does. Compare the squared error in the k poorly determined directions and in the rest with the values printed there for 50 epochs at $\lambda = 0$ and at $\lambda = \frac{1}{3}$.
   3. Repeat the unpenalized run at $\eta = 0.01$ and at $\eta = 0.1$ and read the best epoch from `trainer.history`. Explain how it moves, and state what early stopping requires in practice that the penalty does not.

7. code **Duplicated features.** Use the functions `twins`, `mse`, and `ridge` of Section 2.7.3.
   1. Sweep λ over $10^{-8}, 10^{-7}, \ldots, 10$ and plot both ridge weights and both test errors against λ on a logarithmic axis. For which range of λ does ridge do well on both test sets? Relate the two ends of the range to the two eigenvalues of $\frac{1}{n}\mathbf{X}^\top\mathbf{X}$.
   2. Run gradient descent on the unpenalized loss from $\mathbf{w} = \mathbf{0}$ with learning rate 0.1 for 10^2, 10^4, and 10^6 steps. Compare the weights and test errors with those of least squares and ridge, and explain the result with the factor $(1 - \eta\mu_j)^t$ of each eigenvector.
   3. Fit least squares to labels without noise, $y = x_1 + x_2$. What changes, and why?

8. code **More weights than examples.** ⊠ Generate data with 200 features and only 20 training examples, true weights 0.01, bias 0.05, and noise $\sigma = 0.01$, in minibatches of 5.
   1. Compute the eigenvalues of $\frac{1}{n}\mathbf{X}^\top\mathbf{X}$ for the 20 training inputs. How many are zero, and why does gradient descent never change the weights along their eigenvectors?
   2. Train with $\lambda = 0$ and with $\lambda = 3$ for 30 epochs at learning rate 0.01, once from the random initialization of `WeightDecayScratch` and once from $\mathbf{w} = \mathbf{0}$ (pass `sigma=0`).

Before you run, predict the four validation losses from the norm of the initial weights in the undetermined directions; then compare.

3. Compare all four losses with that of a model that predicts b^* everywhere. What did weight decay achieve here, and what could no value of λ achieve?

Chapter 3

Linear Classification

The previous chapter taught a linear model to predict a number, such as the price of a house. Many questions ask for a category instead. A photo in an online clothing store shows a shirt, a coat or a sandal, not 5.5 of anything. We might be tempted to number the categories and regress on the numbers anyway. The model would then treat a name as a quantity, as if a pullover lay halfway between a trouser and a dress. Nor can a single number express doubt, and a classifier that cannot say "probably a shirt, possibly a pullover" is of limited use. How should a model predict a category, and when can we trust its prediction?

Statisticians had a good answer long before anyone trained a neural network. In 1936 Ronald Fisher wanted to tell species of iris apart from four measurements of their flowers, and he did it with a weighted sum of the measurements, a linear score. Softmax regression, the model of this chapter, does much the same. It gives each class its own linear score and turns the scores into a probability distribution over the classes. Pleasantly little else has to change. Maximum likelihood, which turned Gaussian noise into the squared loss in Chapter 2, now yields the cross-entropy loss, and minibatch stochastic gradient descent trains the model as before. The model is worth learning properly: nearly every classifier in this book, up to the largest language models, ends in the same softmax and is trained with the same loss.

Evaluation is where classification really differs from regression. A probability is not yet a decision. Accuracy can flatter a classifier when one class is rare. And a test set vouches only for data that resemble it.

The chapter therefore has two halves. The first four sections build the classifier: we derive the model and its loss (Section 3.1), meet the clothing images on which we try it (Section 3.2), and implement the model twice. The version we write by hand (Section 3.3) trains to about 83% accuracy and shows us, in its mistakes and in its weights, what a linear classifier can and cannot see; it also fails numerically in an instructive way, which the framework's loss (Section 3.4) repairs. The last three sections ask what that 83% is worth. Accuracy hides a great deal, and the predicted probabilities may or may not mean what they say (Section 3.5). A number measured on 10,000 images has an error bar, which shrinks when the images are fresh and widens when the same test set is consulted again and again (Chapter 3.6). And the data at deployment differ from the training data, as to some degree they always do (Section 3.7).

Throughout the chapter, each example belongs to exactly one class, and the classes have

no order. The next chapter keeps the loss and replaces the linear model by a multilayer network.

Resources and Further Reading

The statistical texts listed with Chapter 2 cover classification as well; the resources below add what is specific to this chapter, and each names the section it accompanies. All are freely accessible online except where noted.

Books and notes

- CS229 Machine Learning lecture notes — Andrew Ng & Tengyu Ma (Stanford) — free PDF; derives logistic and softmax regression and the cross-entropy loss from maximum likelihood, the route of Section 3.1.
- Information theory in this book's mathematics appendix (Chapter 37) — entropy, cross-entropy, and the KL divergence with proofs, the background of the cross-entropy loss in Section 3.1.
- Patterns, Predictions, and Actions — Moritz Hardt & Benjamin Recht — free online; a modern text on prediction whose chapters on generalization, datasets, and deployment accompany Chapter 3.6 and Section 3.7.
- Understanding Machine Learning: From Theory to Algorithms — Shai Shalev-Shwartz & Shai Ben-David — free PDF; uniform convergence and the VC dimension with full proofs (Chapter 3.6).
- Dataset Shift in Machine Learning — Quiñonero-Candela, Sugiyama, Schwaighofer & Lawrence (eds.), MIT Press — the standard reference on covariate, label, and concept shift (Section 3.7; an open-access edition is available from MIT Press).
- Machine Learning in Non-Stationary Environments: Introduction to Covariate Shift Adaptation — Sugiyama & Kawanabe, MIT Press — importance weighting for covariate shift in depth (Section 3.7; print only).

Papers

- Fashion-MNIST: a Novel Image Dataset for Benchmarking Machine Learning Algorithms — Xiao, Rasul & Vollgraf (2017) — the dataset of Section 3.2 and of the rest of the chapter.
- On Calibration of Modern Neural Networks — Guo, Pleiss, Sun & Weinberger (2017) — when a classifier's probabilities can be read as probabilities, and how temperature scaling repairs them (Section 3.1, Section 3.5).
- WILDS: A Benchmark of in-the-Wild Distribution Shifts — Koh, Sagawa, Marklund et al. (2021) — ten datasets with real shifts between hospitals, cameras, regions, and time, each with the in-distribution and shifted test accuracy of standard models (Section 3.7).
- Do ImageNet Classifiers Generalize to ImageNet? — Recht, Roelofs, Schmidt & Shankar (2019) — rebuilds a test set from scratch and finds accuracy drops that a slight distribution shift explains better than test-set reuse (Chapter 3.6, Section 3.7).

3.1 Softmax Regression

The linear regression of Section 2.1 predicts a number, such as the price of a house. Many questions ask instead to which of several categories, or classes, an example belongs. Prob-

lems of this kind are called *classification*. A spam filter chooses between two classes, a diagnostic system among a handful of diseases, and a language model, which predicts the next word, among the tens of thousands of words of its vocabulary. How can a linear model answer such a question, and which loss should train it?

Consider an online store that files each product photo under one of ten categories: T-shirt, trouser, pullover, dress, coat, sandal, shirt, sneaker, bag, or ankle boot. Nothing stops us from coding the categories as 1 to 10 and reusing the linear regression we already have. The trouble is that the codes are names, not quantities. A prediction of 5.5 would place a photo halfway between a coat and a sandal, whatever that might be. Worse, the squared loss would charge 81 times more for mistaking a T-shirt (code 1) for an ankle boot (code 10) than for a trouser (code 2), although nothing makes the first mistake worse. Usually we also want more than a decision. A spam filter should remove an email from the inbox only when it is nearly certain, and a doctor needs to know how likely a diagnosis is, not just which one is most likely. So the model should output a probability for every class.

Softmax regression computes one linear score per class and turns the scores into probabilities. To train it, we reuse the recipe of Section 2.1: describe how the labels are generated and minimize the negative log-likelihood of the observed labels. There, Gaussian noise gave the squared loss. Here, a categorical model of the label gives the cross-entropy loss. Throughout the chapter, each example belongs to exactly one class and the classes have no order. The following sections train the model on images of clothing and ask how well it predicts images that it has not seen.

3.1.1 The Model

3.1.1.1 Class Labels

We start with a problem small enough to write out in full. Each input is a 2×2 grayscale image, so it has four features, the pixel values x_1, x_2, x_3, x_4, and each image shows a cat, a chicken, or a dog. We number the classes 1, 2, 3 in this order. In general there are q classes, and the label y of an example is its class, an integer in $\{1, \dots, q\}$.

Regressing on these integers would impose an order that the classes do not have, as the clothing codes showed. Instead we encode the label as a *one-hot vector* $\mathbf{y} \in \{0, 1\}^q$, which has a 1 in the position of the true class and 0 everywhere else. A chicken is $\mathbf{y} = (0, 1, 0)$, and in general entry y_j is 1 if the example belongs to class j and 0 otherwise. Every class gets its own coordinate, and any two classes are equally far apart. The one-hot vector is also a probability distribution over the classes, the one that puts all its mass on the true class, so we can compare it directly with the probabilities that the model predicts. In formulas, the class y serves as an index and the one-hot vector $\mathbf{y}$ as a vector over the classes. In code, a label is stored as the integer index of its class, counting from 0 as Python does.

3.1.1.2 Linear Scores

Softmax regression gives every class its own linear model. Class j receives a score o_j, a weighted sum of the features plus a bias, with weights and a bias of its own. We write w_{kj} for the weight of feature k in the score of class j. For the 2×2 images the three scores are

$$o_1 = x_1 w_{11} + x_2 w_{21} + x_3 w_{31} + x_4 w_{41} + b_1,$$
$$o_2 = x_1 w_{12} + x_2 w_{22} + x_3 w_{32} + x_4 w_{42} + b_2,$$
$$o_3 = x_1 w_{13} + x_2 w_{23} + x_3 w_{33} + x_4 w_{43} + b_3.$$

Each line is a linear regression of the kind we met in Section 2.1, so the model has 12 weights and 3 biases. A larger score will mean a more probable class. The scores o_j are called *logits*, a name explained below. Drawn as a network (Figure 3.1), the model has one output unit per class, and every output depends on every input: it is a single fully connected layer, like linear regression, but with q outputs instead of one.

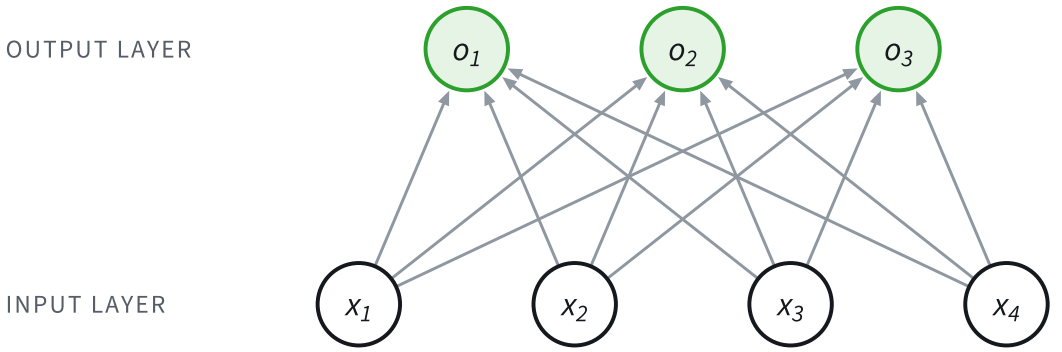


Figure 3.1: Softmax regression as a network with one fully connected layer. Each input x_k feeds each output o_j through its own weight w_{kj}; the weights into o_1 form the first column of $\mathbf{W}$.

We collect the weights in a matrix $\mathbf{W} \in \mathbb{R}^{4\times 3}$, with one row per feature and one column per class, and the biases in a vector $\mathbf{b} \in \mathbb{R}^3$. Column j of $\mathbf{W}$, which we write $\mathbf{w}_j$, holds the weights of class j, so that $o_j = \mathbf{w}_j^\top \mathbf{x} + b_j$. All the scores together are

$$\mathbf{o} = \mathbf{W}^\top \mathbf{x} + \mathbf{b}. \tag{3.1}$$

Entry j of $\mathbf{W}^\top \mathbf{x}$ is the dot product of column $\mathbf{w}_j$ with $\mathbf{x}$, the weighted sum for class j. With d features and q classes, $\mathbf{W} \in \mathbb{R}^{d\times q}$ and $\mathbf{o}, \mathbf{b} \in \mathbb{R}^q$. The layout is a convention and nothing more: storing the features along the rows of $\mathbf{W}$ matches the minibatch form below and the code of this chapter. For the 28×28 clothing images that we will classify, $d = 784$ and $q = 10$, so the model has $7{,}840$ weights and 10 biases.

3.1.1.3 The Softmax

Suppose that for some image the model computes the logits $\mathbf{o} = (1.0, 2.2, 0.3)$. The chicken has the largest score, but the scores are not probabilities: they sum to 3.5, not 1, and nothing prevents a score from being negative or from exceeding 1. Could we not simply fit the scores to the one-hot labels by least squares, as if this were regression with q outputs? Unfortunately not. A linear function of $\mathbf{x}$ grows without bound, so for inputs far from the training data it would predict "probabilities" above 1 or below 0.

We need a map from any vector of q real scores to a probability distribution over q classes: q nonnegative numbers that sum to 1, with a larger score giving a larger probability. Two steps

achieve this. Exponentiating each score makes it positive and keeps the order, and dividing by the sum makes the results sum to 1. Why the exponential, when any positive, increasing function would pass this test? Because it turns sums into products. Adding a constant to a score multiplies its exponential by a constant, so differences of scores become ratios of probabilities, and we will see that the gradient of the loss then takes the simplest form one could hope for. The two steps together form the *softmax* function,

$$\hat{\mathbf{y}} = \mathrm{softmax}(\mathbf{o}) \quad \text{where} \quad \hat{y}_j = \frac{\exp(o_j)}{\sum_{k=1}^{q} \exp(o_k)}. \tag{3.2}$$

Each probability is proportional to the exponential of its score. For $\mathbf{o} = (1.0, 2.2, 0.3)$ the exponentials are $(2.72, 9.03, 1.35)$, their sum is 13.09, and $\hat{\mathbf{y}} = (0.21, 0.69, 0.10)$: the model gives the chicken a probability of 0.69. Since the exponential is increasing, the softmax keeps the order of the scores, and the most probable class is the one with the largest logit,

$$\operatorname*{argmax}_j \hat{y}_j = \operatorname*{argmax}_j o_j.$$

To predict a class, we can skip the softmax and read off the largest logit.

Let us look more closely at what the exponential has given us. Only the differences between logits matter. Adding the same constant c to every logit multiplies the numerator and the denominator of Equation 3.2 by $\exp(c)$, which cancels, so $\mathrm{softmax}(\mathbf{o} + c\mathbf{1}) = \mathrm{softmax}(\mathbf{o})$. Equivalently, the difference of two logits is the logarithm of the ratio of their probabilities,

$$o_i - o_j = \log \frac{\hat{y}_i}{\hat{y}_j}. \tag{3.3}$$

In the example, the logit of the chicken exceeds that of the cat by 1.2, so the chicken is $e^{1.2} \approx 3.3$ times as probable as the cat $(0.69/0.21)$. The q logits therefore carry one redundant degree of freedom. We could fix the score of one class at 0 without changing any prediction, but we keep all q so that every class is treated alike. The redundancy has a practical use as well: subtracting the largest logit before exponentiating keeps exp from overflowing in floating point (Section 3.3).

For two classes, the probabilities depend only on the difference $o = o_1 - o_2$:

$$\hat{y}_1 = \frac{\exp(o_1)}{\exp(o_1) + \exp(o_2)} = \frac{1}{1 + \exp(-o)} = \sigma(o). \tag{3.4}$$

The function σ is the *logistic sigmoid*, which maps the real line onto the interval $(0, 1)$. Softmax regression with two classes is thus nothing new. It is logistic regression, the standard model for binary classification in statistics. As we just saw, $o = \log(\hat{y}_1/\hat{y}_2)$ is the logarithm of the odds of class 1, which statisticians call its logit. Machine learning has extended the name, somewhat loosely, to the scores of any classifier.

3.1.1.4 Minibatches

As in linear regression, we compute the predictions for a minibatch of examples with one matrix product. We stack the n examples of a minibatch as the rows of $\mathbf{X} \in \mathbb{R}^{n \times d}$. Then

$$\begin{aligned} \mathbf{O} &= \mathbf{X}\mathbf{W} + \mathbf{b}, \\ \hat{\mathbf{Y}} &= \mathrm{softmax}(\mathbf{O}), \end{aligned} \tag{3.5}$$

where row i of $\mathbf{O} \in \mathbb{R}^{n \times q}$ holds the logits of example i and the bias vector is added to every row by broadcasting. The softmax acts *rowwise*: it exponentiates each row of $\mathbf{O}$ and divides it by its own sum, so row i of $\hat{\mathbf{Y}}$ is the predicted distribution for example i. Most of the work is the product $\mathbf{X}\mathbf{W}$. For a minibatch of 256 clothing images it multiplies a 256×784 matrix by a 784×10 matrix, about two million multiply-adds, in a single call.

3.1.1.5 Decision Regions

Since the predicted class has the largest logit and every logit is affine in $\mathbf{x}$, the decisions of the model have a simple geometry (Figure 3.2). Class j is predicted wherever $o_j \geq o_k$ for every other class k. Each of these $q - 1$ conditions, $(\mathbf{w}_j - \mathbf{w}_k)^\top \mathbf{x} + b_j - b_k \geq 0$, is a linear inequality that holds on one side of a hyperplane. The region of class j is the intersection of these half-spaces, a convex polyhedron, and two neighboring regions meet on the hyperplane where their scores tie.

With two classes, $\hat{y}_1 = \sigma(o)$ depends on $\mathbf{x}$ only through $o = (\mathbf{w}_1 - \mathbf{w}_2)^\top \mathbf{x} + b_1 - b_2$. Inputs with the same predicted probability lie on a hyperplane perpendicular to $\mathbf{w}_1 - \mathbf{w}_2$, and the decision boundary is the hyperplane on which $\hat{y}_1 = \frac{1}{2}$. The softmax bends the probabilities, but the boundaries between classes stay flat. This is a real limitation. Four points at the corners of a square, with each class occupying two opposite corners, cannot be separated by any line. Such classes are beyond the reach of this model, and remain so until Chapter 4 adds hidden layers.

Note, however, that all of this rests on the assumption we made at the start: every example belongs to exactly one class, and the classes have no order. Two common kinds of problems do not satisfy it. In *multi-label classification*, several labels can apply to the same example, as when a news article covers both business and space flight. Such problems use a separate sigmoid for every label instead of one softmax, because the labels do not compete for probability (Tsoumakas and Katakis 2007). In ordinal regression, the classes are ordered, as age groups from infant to senior are, and a loss that ignores the order discards information (Moon et al. 2010).

3.1.2 The Cross-Entropy Loss

We now need a loss that measures how well the predicted probabilities $\hat{\mathbf{y}}$ fit the labels. We could invent one, but there is no need. The squared loss of Section 2.1 was not chosen for convenience alone: it fell out of a model of how the labels arise, as the negative log-likelihood of the observed labels. Let us see what the same recipe produces for class labels.

3.1.2.1 Maximum Likelihood

The softmax output is a probability distribution over the classes, so it defines a model of the labels: given features $\mathbf{x}$, an example belongs to class j with probability $\hat{y}_j$. The probability

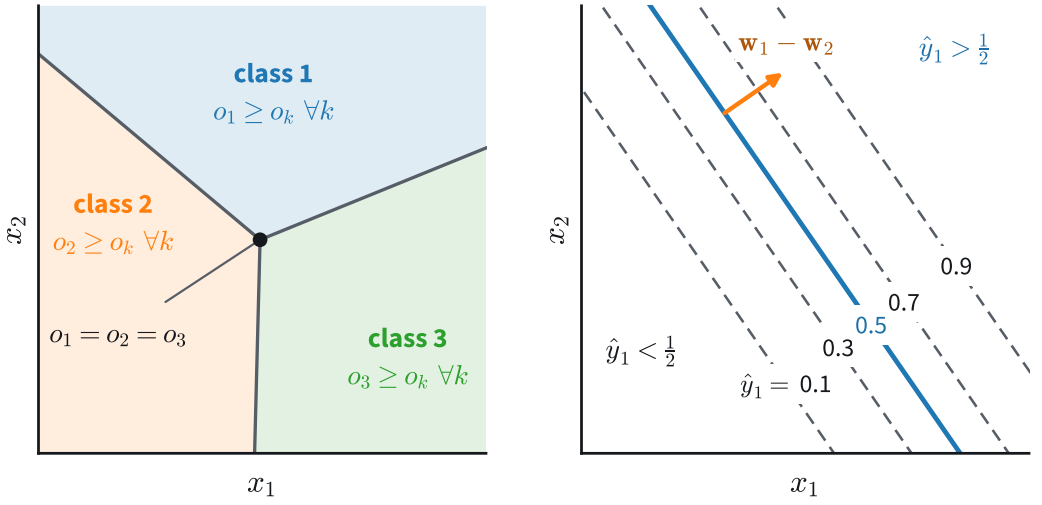


Figure 3.2: Decision regions of softmax regression in the plane. Left: with three classes, the plane splits into three convex regions, one per class, bounded by the lines on which two scores tie; the three lines meet where all scores are equal. Right: with two classes, inputs of equal predicted probability $\hat{y}_1 = \sigma(o)$ lie on parallel lines perpendicular to $\mathbf{w}_1 - \mathbf{w}_2$, and the decision boundary is the line $\hat{y}_1 = \frac{1}{2}$.

of the observed label y is $\hat{y}_y$, the entry of $\hat{\mathbf{y}}$ at the true class. With the one-hot vector we can write it as a product over all classes,

$$P(y \mid \mathbf{x}) = \hat{y}_y = \prod_{j=1}^{q} \hat{y}_j^{y_j},$$

since every factor other than the one for the true class is raised to the power 0. This is the *categorical distribution*, which plays the role that the Gaussian noise model played for linear regression.

If the labels of different examples are drawn independently, the probability of all n labels is the product of these probabilities. As in linear regression, we take the negative logarithm, which turns the product into a sum and the maximization into a minimization:

$$-\log \prod_{i=1}^{n} P(y^{(i)} \mid \mathbf{x}^{(i)}) = \sum_{i=1}^{n} l(\mathbf{y}^{(i)}, \hat{\mathbf{y}}^{(i)}), \tag{3.6}$$

where the loss of one example is

$$l(\mathbf{y}, \hat{\mathbf{y}}) = -\sum_{j=1}^{q} y_j \log \hat{y}_j = -\log \hat{y}_y. \tag{3.7}$$

The loss is minus the logarithm of the probability that the model assigns to the true class. It is called the *cross-entropy loss*, for a reason we give below. For the logits $(1.0, 2.2, 0.3)$ and an

image of a chicken, the loss is $-\log 0.69 \approx 0.37$. Had the image shown a dog, to which the model gives probability 0.10, the loss would be $-\log 0.10 \approx 2.3$, about six times as much.

Since every $\hat{y}_j$ lies between 0 and 1, the loss is never negative. It is 0 only if the model gives the true class probability 1, which finite logits never do, since every exponential is positive. At the other extreme, a confident mistake costs without limit: as $\hat{y}_y \to 0$, the loss grows to infinity.

3.1.2.2 The Gradient

Training by minibatch SGD needs the gradient of the loss with respect to the parameters. We first differentiate with respect to the logits. Substituting the softmax Equation 3.2 into the loss and using $\sum_j y_j = 1$, we obtain

$$\begin{aligned} l(\mathbf{y}, \hat{\mathbf{y}}) &= -\sum_{j=1}^{q} y_j \log \frac{\exp(o_j)}{\sum_{k=1}^{q} \exp(o_k)} \\ &= \sum_{j=1}^{q} y_j \log \sum_{k=1}^{q} \exp(o_k) - \sum_{j=1}^{q} y_j o_j \\ &= \log \sum_{k=1}^{q} \exp(o_k) - o_y. \end{aligned}$$

The loss is the *log-sum-exp* of all logits minus the logit of the true class. Log-sum-exp is a smooth version of the maximum: it exceeds the largest logit by at most $\log q$ (exercise 5), so the loss is small when the true class leads all others by a wide margin. The derivative of log-sum-exp with respect to o_j is $\exp(o_j)/\sum_k \exp(o_k)$, the softmax itself. Therefore

$$\frac{\partial l}{\partial o_j} = \text{softmax}(\mathbf{o})_j - y_j, \quad \text{that is,} \quad \nabla_{\mathbf{o}}\, l = \hat{\mathbf{y}} - \mathbf{y}. \tag{3.8}$$

This is the simplicity that the exponential promised: the gradient is the predicted distribution minus the observed one. For the chicken it is $(0.21, 0.69, 0.10) - (0, 1, 0) = (0.21, -0.31, 0.10)$. A gradient step raises the logit of the true class and lowers the others, each in proportion to the probability that it received by mistake. The entries sum to zero, as they must, since adding a constant to all logits leaves the loss unchanged.

Since $o_j = \mathbf{w}_j^\top \mathbf{x} + b_j$, the chain rule gives the gradients for the parameters, $\partial l/\partial \mathbf{w}_j = (\hat{y}_j - y_j)\,\mathbf{x}$ and $\partial l/\partial b_j = \hat{y}_j - y_j$. A step moves the weights of the true class toward $\mathbf{x}$ and those of every other class away from it. For a minibatch whose one-hot labels are the rows of $\mathbf{Y} \in \{0,1\}^{n \times q}$, the gradient of the average loss with respect to $\mathbf{W}$ is $\frac{1}{n}\mathbf{X}^\top(\hat{\mathbf{Y}} - \mathbf{Y})$, a $d \times q$ matrix like $\mathbf{W}$.

Linear regression has a gradient of the same form: the derivative of $\frac{1}{2}(\hat{y} - y)^2$ with respect to the prediction is the error $\hat{y} - y$. This is no coincidence, and Section 36.4.3.3 explains it. It is instructive to set the two models side by side:

	Linear regression	Softmax regression
label	a real number y	a class y, one-hot $\mathbf{y}$

	Linear regression	Softmax regression
prediction	$\hat{y} = \mathbf{w}^\top \mathbf{x} + b$	$\hat{\mathbf{y}} = \text{softmax}(\mathbf{W}^\top \mathbf{x} + \mathbf{b})$
model of the label	Gaussian centered at $\hat{y}$	categorical with probabilities $\hat{\mathbf{y}}$
loss (negative log-likelihood)	$\frac{1}{2}(\hat{y} - y)^2$	$-\log \hat{y}_y$
gradient	$\hat{y} - y$ with respect to $\hat{y}$	$\hat{\mathbf{y}} - \mathbf{y}$ with respect to $\mathbf{o}$

The loss is also convex in the logits. With two classes this is easy to check: with $o = o_1 - o_2$ as before, the loss of an example of class 1 is $\log(1 + \exp(-o))$, and its second derivative is $\sigma(o)(1 - \sigma(o)) > 0$, the variance of a coin that shows heads with probability $\sigma(o)$. With q classes the variance becomes the covariance matrix of the predicted distribution, which is positive semidefinite (exercise 3). Since the logits are affine in $\mathbf{W}$ and $\mathbf{b}$, the training loss is convex in the parameters, and every local minimum is a global one. Gradient descent cannot get stuck in a poor one, a comfort that the deeper models of later chapters do not offer.

3.1.2.3 Entropy and Cross-Entropy

The name of the loss comes from information theory, which measures information by surprise. An outcome of probability p carries the *surprisal* $-\log p$. An outcome that was certain ($p = 1$) does not surprise us at all, and the rarer an outcome, the larger its surprisal. Our loss is the surprisal of the true label under the prediction of the model.

Suppose that outcomes are drawn from a distribution $\mathbf{p}$ over q classes. The *entropy* of $\mathbf{p}$ is the expected surprisal of an observer who knows $\mathbf{p}$,

$$H(\mathbf{p}) = -\sum_{j=1}^{q} p_j \log p_j. \tag{3.9}$$

It is 0 when one outcome is certain and largest, $\log q$, when all outcomes are equally likely. With the natural logarithm it is measured in *nats*; with $\log_2$ it is measured in bits, and one nat is $1/\log 2 \approx 1.44$ bits.

An observer who predicts the same outcomes with a distribution $\hat{\mathbf{p}}$ instead has the expected surprisal

$$H(\mathbf{p}, \hat{\mathbf{p}}) = -\sum_{j=1}^{q} p_j \log \hat{p}_j, \tag{3.10}$$

the *cross-entropy* of $\hat{\mathbf{p}}$ relative to $\mathbf{p}$. The loss Equation 3.7 is the cross-entropy $H(\mathbf{y}, \hat{\mathbf{y}})$ between the one-hot label and the prediction, hence its name. One fact about these quantities tells us what minimizing the loss achieves: predicting with the wrong distribution can only increase the average surprise,

$$H(\mathbf{p}, \hat{\mathbf{p}}) \geq H(\mathbf{p}), \quad \text{with equality if and only if } \hat{\mathbf{p}} = \mathbf{p}. \tag{3.11}$$

This is Gibbs' inequality, proved in Section 37.1. That the entropy is also the average length of the shortest code for the outcomes is shown in Section 37.2.

Labels are often uncertain even when the features are known. A small, blurry photo may show a shirt with probability 0.6 and a T-shirt with probability 0.4. If photos like it carry these labels at random according to $\mathbf{p} = (0.6, 0.4)$, the expected loss of a prediction $\hat{\mathbf{y}}$ is $H(\mathbf{p}, \hat{\mathbf{y}})$. Predicting $(0.6, 0.4)$ costs $H(\mathbf{p}) = 0.67$ nats on average, while the more confident $(0.9, 0.1)$ costs 0.98 nats. By Gibbs' inequality, the expected loss is smallest when the model reports the true probabilities, so cross-entropy rewards honest probabilities rather than exaggerated ones. For the same reason the loss accepts labels that are themselves distributions, such as the fractions of annotators who chose each class: the first form of Equation 3.7, $-\sum_j y_j \log \hat{y}_j$, applies unchanged with a probability vector in place of the one-hot $\mathbf{y}$ (the second form, $-\log \hat{y}_y$, needs a one-hot label).

3.1.3 Accuracy and Confidence

A classifier is usually judged by its *accuracy*, the fraction of examples on which the class with the largest logit is the true class, or by its error rate, one minus the accuracy. Yet accuracy is not what we train for. We minimize cross-entropy, and it is worth understanding why, and what this choice does to the confidence of the model.

3.1.3.1 The Error Rate

Why not minimize the number of training errors directly? Whether an example is classified correctly depends on its *margin*, the logit of the true class minus the largest other logit,

$$m = o_y - \max_{j \neq y} o_j. \tag{3.12}$$

The example is classified correctly if $m > 0$ and wrongly if $m < 0$. Its error, 1 for a mistake and 0 otherwise, is a step function of the margin (Figure 3.3). Wherever the step has a derivative, the derivative is zero. A small change of the weights either changes no decision, so the number of errors stays the same, or it flips a decision, so the number jumps. Gradient descent gets no direction from it.

Cross-entropy is a smooth function of the margin. With two classes, the probability of the true class is $\sigma(m)$ by Equation 3.4, so the loss is

$$l = -\log \sigma(m) = \log\left(1 + \exp(-m)\right). \tag{3.13}$$

With q classes, the loss $\log \sum_k \exp(o_k) - o_y = \log\left(1 + \sum_{j \neq y} \exp(o_j - o_y)\right)$ is at least as large, since the sum contains the term $\exp(-m)$. Figure 3.3 shows the two-class loss. It decreases everywhere, so every example contributes a gradient that pushes its margin up, and the misclassified examples push hardest. It also lies above the error: at $m \leq 0$ the loss is at least $\log 2$, so the sum of the losses divided by $\log 2$ bounds the number of training errors. A model with an average training loss of 0.07 nats misclassifies at most $0.07/\log 2 \approx 10\%$ of its training examples. The bound is crude, though. At an average loss of 0.45 nats it allows 65% errors, and a model with such a loss typically makes far fewer, since the examples that it classifies correctly contribute loss as well. What the bound does

guarantee is that driving the loss toward zero cannot leave the error rate high, and the loss, unlike the step, is a function that gradient descent can follow.

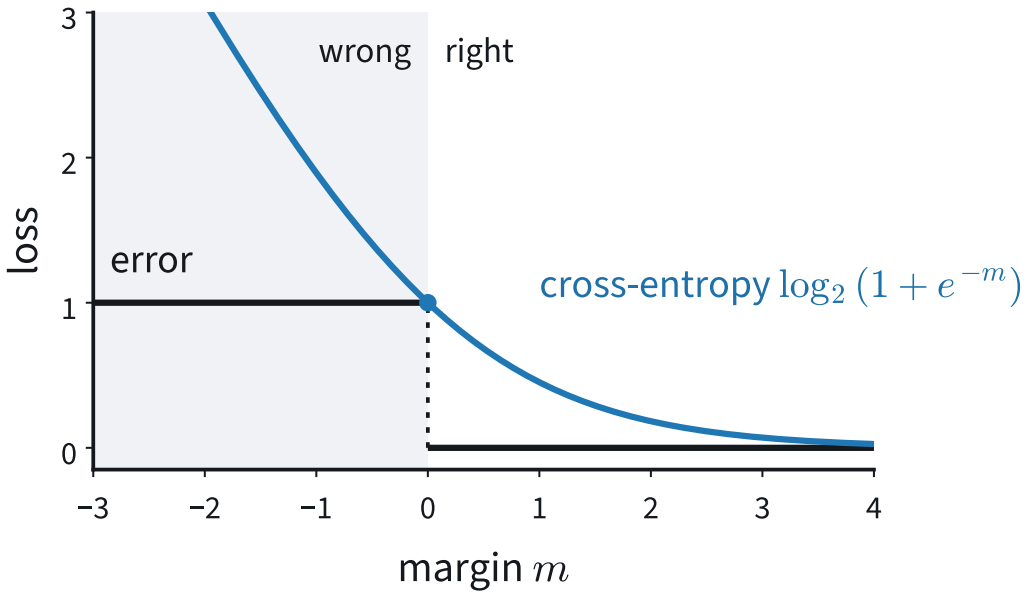


Figure 3.3: The error and the cross-entropy loss of an example as functions of its margin m, the logit of the true class minus the largest other logit. The error is a step: 1 for a misclassified example ($m < 0$) and 0 otherwise. The two-class cross-entropy $\log(1 + e^{-m})$, divided by $\log 2$ so that it is measured in bits, is smooth and decreasing, equals 1 at $m = 0$, and lies above the error everywhere.

The loss never reaches zero. It keeps decreasing after the margin turns positive, so it keeps rewarding the model for raising its confidence in decisions that are already right. Raising the probability of the true class from 0.51 to 0.99 changes no decision, but lowers the loss from 0.67 to 0.01.

3.1.3.2 Calibration and Temperature

Taken to its limit, this reward for confidence has a curious consequence. Suppose that some parameters classify every training example correctly. Multiplying $\mathbf{W}$ and $\mathbf{b}$ by a constant larger than 1 multiplies every margin by it and lowers every loss, so the training loss has no minimum, and gradient descent keeps growing the weights (exercise 7). The decisions stay as they were. Only the probabilities move, toward 0 and 1.

The scale of the logits thus sets the confidence of a prediction and nothing else. Dividing all logits by a *temperature* $T > 0$ before the softmax,

$$\hat{\mathbf{y}} = \text{softmax}(\mathbf{o}/T), \tag{3.14}$$

sharpens the distribution for $T < 1$ and flattens it for $T > 1$ (Figure 3.4), but it keeps the order of the logits, so the predicted class, and with it the accuracy, does not change.

The name comes from statistical physics, where the same formula gives the occupation of energy states at temperature T (Section 36.4.4).

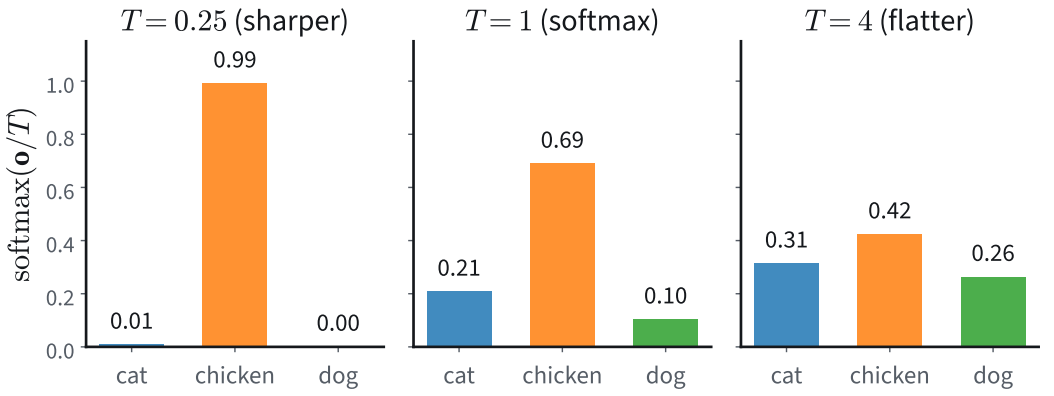


Figure 3.4: Softmax probabilities of the logits $\mathbf{o} = (1.0, 2.2, 0.3)$ for cat, chicken, and dog at three temperatures. At $T = 0.25$ nearly all probability goes to the chicken, $T = 1$ is the plain softmax, and at $T = 4$ the distribution is close to uniform. The order of the bars is the same in all three panels.

A low loss, then, does not certify the probabilities. Whether a prediction made with confidence 0.9 is right nine times out of ten is something we have to measure, and Section 3.5 measures it for the model we are about to train. Nor does a low training loss tell us how well the model classifies images that it has not seen. For that we need data.

3.1.4 Summary

Softmax regression computes one linear score, or logit, per class, $\mathbf{o} = \mathbf{W}^\top \mathbf{x} + \mathbf{b}$, and turns the scores into probabilities with the softmax, which exponentiates and normalizes them; only differences between logits matter, and the boundaries between classes are hyperplanes. Reading the output as a categorical distribution of the label and maximizing the likelihood of the observed labels gives the cross-entropy loss $-\log \hat{y}_y$, just as Gaussian noise gave the squared loss for regression. The loss is convex, and its gradient with respect to the logits is again prediction minus label, $\hat{\mathbf{y}} - \mathbf{y}$. It bounds the number of errors, if loosely, by a function that gradient descent can follow, and it keeps rewarding confidence in decisions that are already right, so the probabilities of a trained model have to be checked before they are trusted.

3.1.5 Exercises

1. **By hand.** A three-class model computes the logits $(2, 0, -1)$ for an input.
    1. Compute the softmax probabilities, and the loss and the gradient $\hat{\mathbf{y}} - \mathbf{y}$ for each of the three possible labels. Check that each gradient sums to zero.
    2. Subtract 5 from every logit and repeat. State in one sentence what changed and what did not.
    3. Divide the logits by $T = 2$ and by $T = \frac{1}{2}$. Which quantities change, and which class is predicted in each case?

2. **Softmax and sigmoid.** By Equation 3.4, logistic regression is softmax regression with two classes and one redundant logit removed. A colleague hands you a fitted logistic model $P(y = 1 \mid x) = \sigma(\hat{\beta}_0 + \hat{\beta}_1 x)$ with $\hat{\beta}_0 = 2$ and $\hat{\beta}_1 = -1$.

   1. Characterize all two-class softmax coefficients (w_1, b_1) and (w_2, b_2) whose predictions coincide with the logistic model for every x, and identify the pair with $o_2 \equiv 0$.
   2. Show that gradient descent on the cross-entropy loss of the two-class softmax model never changes $w_1 + w_2$ or $b_1 + b_2$. Suppose that gradient descent starts from $w_1 = w_2 = b_1 = b_2 = 0$ on the colleague's training data and converges to coefficients that reproduce the logistic model. Determine which pair it reaches.

   Adapted from James, Witten, Hastie, and Tibshirani, An Introduction to Statistical Learning, Exercise 4.12.

3. **Softmax Hessian.** As in Section 3.1.2.2, write the cross-entropy loss as a function of the logits, $l(\mathbf{y}, \mathbf{o}) = g(\mathbf{o}) - \mathbf{y}^\top \mathbf{o}$ with $g(\mathbf{o}) = \log \sum_k \exp(o_k)$.

   1. Compute the Hessian $\partial^2_{\mathbf{o}}\, l(\mathbf{y}, \mathbf{o})$ and show that it does not depend on $\mathbf{y}$.
   2. Let $\mathbf{e}_Y$ be the one-hot vector of a class Y drawn from softmax$(\mathbf{o})$. Compute the covariance matrix of $\mathbf{e}_Y$ and show that it equals the Hessian from the previous step.
   3. Conclude that g is convex and that the cross-entropy loss is convex in the logits.

4. **Bradley–Terry model.** The Bradley–Terry model (Bradley and Terry 1952) uses a logistic model to capture preferences. A user who chooses between apples and oranges picks each item with the probability given by the softmax of two scores, o_{apple} and o_{orange}. Now add a default option of choosing neither, with a fixed score $o_{\text{none}} = 0$.

   1. Determine whether adding the same constant to o_{apple} and o_{orange} changes the choice probabilities, first without and then with the default option. Conclude which scores the observed choices can determine in each model.
   2. Compare the ratio $P(\text{apple})/P(\text{orange})$ with and without the default option. Describe a choice situation in which real users would not keep this ratio fixed.

5. **Log-sum-exp.** The function $\text{lse}(a, b) = \log(\exp(a) + \exp(b))$ is the log-sum-exp of Section 3.1.2.2 with two arguments, so its gradient is the softmax of (a, b).

   1. Prove that
   $$\max(a, b) < \text{lse}(a, b) \le \max(a, b) + \log 2, \tag{3.15}$$
   with equality on the right exactly when $a = b$, and show that the gap to $\max(a, b)$ can be made arbitrarily small. Hint: without loss of generality set $b = 0$ and $a \ge 0$.
   2. Prove that for every $\lambda > 0$
   $$\max(a, b) < \lambda^{-1}\text{lse}(\lambda a, \lambda b) \le \max(a, b) + \lambda^{-1} \log 2, \tag{3.16}$$
   and conclude that $\lambda^{-1}\text{lse}(\lambda a, \lambda b) \to \max(a, b)$ as $\lambda \to \infty$.
   3. Construct an analogous smooth minimum.
   4. Extend the definition and both bounds to q numbers, showing that the gap to $\max_i a_i$ is at most $\log q$.

6. **Temperature.** Let $P = \text{softmax}(\mathbf{o})$, and let the distribution at temperature T be softmax$(\mathbf{o}/T)$. Suppose we pick another distribution Q with $Q(i) \propto P(i)^\alpha$ for $\alpha > 0$.

1. Which choice of α corresponds to doubling the temperature? Which choice corresponds to halving it?
2. Suppose that exactly s of the q logits attain the largest value. Find the limit of $\mathrm{softmax}(\mathbf{o}/T)$ as $T \to 0$.
3. Let $\bar{o} = \frac{1}{q}\sum_j o_j$ be the mean logit. Show that for large T

$$\mathrm{softmax}(\mathbf{o}/T)_i = \frac{1}{q} + \frac{o_i - \bar{o}}{qT} + O(T^{-2}). \tag{3.17}$$

4. Temperature scaling fits a single T by minimizing the cross-entropy on a held-out set with the logits of the classifier held fixed. Show that this cross-entropy is a convex function of $\beta = 1/T$, so that the fit is a one-dimensional convex problem. Hint: use the convexity of g from exercise 3.

7. **Separable data.** ▣ Suppose that some parameters $(\mathbf{W}, \mathbf{b})$ classify every training example correctly and without ties: on each example, the logit of the true class strictly exceeds every other logit. Consider the scaled parameters $(c\mathbf{W}, c\mathbf{b})$ for $c > 0$.
   1. Show that the cross-entropy loss of every training example strictly decreases as c grows and tends to 0 as $c \to \infty$.
   2. Conclude that the training loss has no minimizer and that every sequence of parameters whose training loss tends to 0 has unbounded norm.
   3. Add the weight decay penalty of Section 2.7, applied to both weights and biases, $\frac{\lambda}{2}(\|\mathbf{W}\|^2 + \|\mathbf{b}\|^2)$ with $\lambda > 0$, to the training loss. Show that the penalized loss has a minimizer.

8. **Squared error on probabilities.** Instead of cross-entropy, we could fit the predicted probabilities to the one-hot label with the squared loss $\frac{1}{2}\|\hat{\mathbf{y}} - \mathbf{y}\|^2$.
   1. Show that the derivatives of the softmax are $\partial \hat{y}_k / \partial o_j = \hat{y}_k(\mathbf{1}(j = k) - \hat{y}_j)$, and use them to compute the gradient of the squared loss with respect to the logits.
   2. For two classes, write both losses as functions of the margin m and compute their derivatives with respect to m. Show that as $m \to -\infty$, that is, for a confident mistake, the derivative of the squared loss tends to 0 while that of the cross-entropy tends to -1.
   3. Explain what this difference means for a model that starts training with confident wrong predictions.

9. code **Gumbel-max trick.** ▣ Suppose that a label is generated by perturbing every logit with independent noise and reporting the class with the largest perturbed score, $y = \mathrm{argmax}_i\,(o_i + \gamma_i)$. Gaussian noise gives the probit model, whose class probabilities have no closed form; one other noise distribution reproduces the softmax probabilities exactly. Let $\gamma_1, \ldots, \gamma_q$ be i.i.d. draws from the standard Gumbel distribution, whose CDF is $P(\gamma \le z) = \exp(-\exp(-z))$.
   1. Show that $P(o_k + \gamma_k \le z) = \exp(-\exp(o_k)\exp(-z))$.
   2. Show that $P\left(\mathrm{argmax}_i\,(o_i + \gamma_i) = k\right) = \mathrm{softmax}(\mathbf{o})_k$. Hint: condition on the value of $o_k + \gamma_k$ and use the substitution $u = \exp(-z)$.
   3. Conclude that scaling the noise, $\gamma_i \mapsto T\gamma_i$, samples from $\mathrm{softmax}(\mathbf{o}/T)$.
   4. Verify your result numerically: for $\mathbf{o} = (1.0, 2.2, 0.3)$ and $T \in \{0.25, 1, 4\}$, draw 10^5 samples of $\mathrm{argmax}_i(o_i + T\gamma_i)$ and confirm that the label frequencies match $\mathrm{softmax}(\mathbf{o}/T)$ up to Monte Carlo error.

3.2 Image Classification Data

The softmax regression of Section 3.1 is ready to be trained. What it lacks is data. Our examples will be images, and the choice of dataset decides what our experiments can show. Which dataset should we use? For the purposes of this book, a good one is hard enough that a better model scores visibly higher, and small enough that a model trains in minutes inside a notebook.

Unfortunately, the two best-known image datasets each meet only one of these needs. MNIST (LeCun, Léon Bottou, et al. 1998), 70,000 grayscale images of handwritten digits with 28×28 pixels in 10 classes, is small but too easy. Convolutional networks classify more than 99% of its test digits correctly, and even classical methods reach about 97%, so models that differ in kind differ in accuracy by a few points at most. ImageNet (Deng et al. 2009), with 1.2 million color images in 1,000 classes, separates models well, but training a network on it takes hours to days of GPU time, a long wait for a notebook cell.

Fashion-MNIST (Xiao et al. 2017) was designed as a harder replacement for MNIST. It keeps the format, 70,000 grayscale images of 28×28 pixels in 10 classes, but shows garments instead of digits, and garments are harder to tell apart: a shirt can look like a T-shirt, a pullover, or a coat. Linear classifiers reach about 83% accuracy on it and good convolutional networks more than 90%, which leaves room to tell models apart. We use it for the rest of this chapter and for much of the book.

In this section we load Fashion-MNIST in each framework, read it in minibatches, check the shape of what the loader delivers, and look at a few images before any model touches them. None of this is deep, but it is worth doing with care, since every later classifier in the book reads its data through the class we write here.

```
import time
import torchvision
from torchvision import transforms
```

```
d2l.use_svg_display()
```

3.2.1 Loading the Dataset

Fashion-MNIST has 7,000 images of each of the ten garments named in Section 3.1, from T-shirt to ankle boot. Its authors split them into 60,000 training and 10,000 test images, 6,000 and 1,000 per class. Each pixel stores a brightness between 0 and 255, with the background at 0.

As in Section 2.4, a `DataModule` holds the data and hands minibatches to the `Trainer` through its methods `train_dataloader` and `val_dataloader`. The class `FashionMNIST` downloads the dataset on first use and keeps both splits in memory. It has two hyperparameters: `batch_size`, the number of examples per minibatch, and `resize`, the height and width of the images that the loader delivers. By default the images keep their native 28×28 pixels; networks in later chapters that are built for larger inputs ask for enlarged images instead.

The torchvision library downloads Fashion-MNIST and serves it as a dataset object. Its `transform` argument resizes each image and converts it into a tensor with values in $[0, 1]$.

```
class FashionMNIST(d2l.DataModule):
    """The Fashion-MNIST dataset."""
    def __init__(self, batch_size=64, resize=(28, 28)):
        super().__init__()
        self.save_hyperparameters()
        trans = transforms.Compose([transforms.Resize(resize),
                                    transforms.ToTensor()])
        self.train = torchvision.datasets.FashionMNIST(
            root=self.root, train=True, transform=trans, download=True)
        self.val = torchvision.datasets.FashionMNIST(
            root=self.root, train=False, transform=trans, download=True)
```

We load the dataset and count the images in each split.

```
data = FashionMNIST()
len(data.train), len(data.val)
```

```
(60000, 10000)
```

There are only two splits: Fashion-MNIST ships no validation set. In this book the 10,000 test images serve as validation data instead, and the `Trainer` reports the loss on them after every epoch. This is convenient, and it is not quite honest. A test set that we consult again and again while we compare models stops being an unbiased test of the model we finally choose. How much harm this does is a question that Chapter 3.6 answers with numbers.

The labels are class indices from 0 to 9. To read them, we map each index to the name of its class.

```
@d2l.add_to_class(FashionMNIST)
def text_labels(self, indices):
    """Return text labels."""
    labels = ['T-shirt', 'trouser', 'pullover', 'dress', 'coat',
              'sandal', 'shirt', 'sneaker', 'bag', 'ankle boot']
    return [labels[int(i)] for i in indices]
```

3.2.2 Reading Minibatches

To a framework, an image is a three-dimensional tensor with c *channels*, one for each color, a height h, and a width w in pixels. Here $c = 1$ because the images are grayscale; color photographs have $c = 3$. Frameworks do not agree on where the channel axis goes: *channel-first* layout stores an image as $c \times h \times w$, *channel-last* layout as $h \times w \times c$.

The loader for the training split draws the examples in a new random order in every epoch, so that the minibatches differ from epoch to epoch; the loader for the validation split reads them in their stored order. Both convert the pixel values from integers between 0 and 255 into floating-point numbers between 0 and 1, and resize the images if asked to. Since 64 does not divide 60,000, the last minibatch of an epoch is smaller than the others (32 images here). Both loaders keep it, so that every epoch covers every image.

`DataLoader` reads the dataset in minibatches. Its `num_workers` processes, 4 by default in `DataModule`, load and transform the next minibatches in parallel while the model works on the

current one.

```
@d2l.add_to_class(FashionMNIST)
def get_dataloader(self, train):
    data = self.train if train else self.val
    return torch.utils.data.DataLoader(data, self.batch_size, shuffle=train,
                                       num_workers=self.num_workers)
```

Which layout does our loader use? We read one minibatch and look at the shape of its first image.

```
X, y = next(iter(data.train_dataloader()))
X[0].shape  # Channel-first: (channels, height, width)
```

```
torch.Size([1, 28, 28])
```

A minibatch adds an axis for the examples in front, and its labels form a vector with one entry per example.

```
print(X.shape, X.dtype, y.shape, y.dtype)
```

```
torch.Size([64, 1, 28, 28]) torch.float32 torch.Size([64]) torch.int64
```

The images arrive as 32-bit floats and the labels as integers. One minibatch of images takes $64 \times 784 \times 4$ bytes, about 200 KB, and the whole training set as floats about 190 MB. By today's standards this is tiny. The dataset fits in memory many times over, so the loader never waits for a disk; its time goes into converting, resizing, and collecting the images.

Is that work fast enough? A loader that is slower than the model leaves the processor waiting for data. We time one pass over the training set without any model.

```
tic = time.time()
for X, y in data.train_dataloader():
    continue
f'{time.time() - tic:.2f} sec'
```

```
'2.24 sec'
```

The number above holds only for the machine that ran this notebook, since it depends on the number of loader workers, the framework, and the hardware. Whatever its value, it is a lower bound: no epoch of training can take less time than reading its data. When training seems slow, this is the first number to check. If an epoch takes barely longer than the loader alone, the model is waiting for data, and more loader workers or prefetching of the next minibatches will help more than any change to the model.

3.2.3 Inspecting the Images

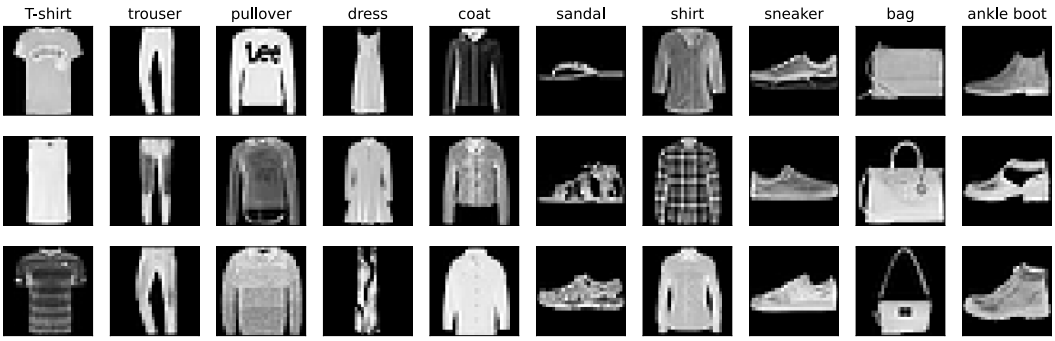
We look at the data before we train on it. A look catches errors that no loss curve reveals: labels attached to the wrong images, images flipped or transposed, a channel axis in the wrong place. A model trains on such data without complaint, only worse. The method `visualize` shows the images of a minibatch, each titled with its label. It calls the library func-

tion `d2l.show_images`, which lays out a list of images in a grid with optional titles. Matplotlib would draw single-channel images in false color, so `visualize` sets the gray color map. Since every image has a single channel, reshaping removes the channel axis in either layout.

```
@d2l.add_to_class(FashionMNIST)
def visualize(self, batch, nrows=1, ncols=8, labels=None):
    """Show the images of a batch in grayscale, titled by their labels."""
    X, y = batch
    if labels is None:
        labels = self.text_labels(y)
    X = d2l.reshape(X, (-1, *self.resize))  # Drop the channel axis
    d2l.show_images(X, nrows, ncols, titles=labels, cmap='gray')
```

We show three images of every class, one class per column: the first three of each class in the validation set, which the validation loader always delivers in the same order.

```
X, y = next(iter(data.val_dataloader()))
labels = d2l.numpy(y)
idx = [int(np.flatnonzero(labels == c)[r]) for r in range(3) for c in range(10)]
titles = data.text_labels(range(10)) + [''] * 20
data.visualize([d2l.stack([X[i] for i in idx]), None], nrows=3, ncols=10,
               labels=titles)
```



The labels match the garments, the garments are upright, and the background is dark, as stored. Note, however, how much some garments of different classes look alike. The T-shirts, pullovers, coats, and shirts in the columns above share an outline and differ in a collar, a button, or the length of a sleeve, details that 28×28 pixels barely resolve; the sandals, sneakers, and ankle boots form a second such family. A trouser or a bag, by contrast, looks like nothing else. We can already guess where a classifier will make most of its mistakes. How many it makes, and whether our guess was right, we learn once we have trained one (Section 3.3).

3.2.4 Summary

Fashion-MNIST has 60,000 training and 10,000 test images of garments in 10 classes, grayscale with 28×28 pixels. It is small enough to train on in minutes, and hard enough that better models score visibly higher, unlike MNIST. The class `FashionMNIST` loads the published files, and its loaders deliver images as floats in $[0, 1]$ and labels as class indices; the training loader reshuffles every epoch, and both loaders keep the last, incomplete minibatch. A

minibatch of images is a four-dimensional tensor, with one axis each for the examples, the channels, the height, and the width. Before training, we look at the images with their labels, and we time the loader, since no epoch can be faster than reading its data.

3.2.5 Exercises

1. code **Class means.** Compute the mean image of each class over the training set, and show the ten means as a grid of images.
   1. Compute the Euclidean distance between every pair of class means and present the distances as a 10×10 table. Which pairs are closest, and which class is farthest from all others?
   2. Predict from the table which classes a linear classifier will confuse most often and which it will almost never confuse. Keep your prediction; Section 3.3 computes the answer.
   3. Name a pair of classes that a person tells apart easily although their means are close, and say which pixels carry the difference.
2. **Shuffling and validation.** `get_dataloader` reshuffles the training split every epoch and reads the validation split in the same order every epoch. Suppose the validation loader, with the default `batch_size=64`, were reshuffled every epoch as well.
   1. Explain whether the validation loss and accuracy, averaged over all 10,000 validation examples, would change.
   2. The `Trainer` weights each validation batch by its number of examples. Suppose it took the unweighted mean of the per-batch values instead. Explain whether that mean would change.
   3. Explain whether the images that `visualize` shows for the first validation batch would change.
3. code **Reshaping versus transposing.** `visualize` removes the channel axis with a reshape, which is correct only because $c = 1$.
   1. A minibatch of color images has shape $(n, 3, h, w)$ in channel-first layout. Show that reshaping it to $(n, h, w, 3)$ does not move the channel axis to the end, by computing which stored value lands at position $(0, 0, 0, 1)$ of the result.
   2. Write the transposition that converts it correctly, apply both versions to a minibatch of 8 random arrays of shape $(3, 4, 5)$, and check which one agrees with indexing the original array.
4. code **Loader throughput.** The time of one pass over the training set depends on the minibatch size and on how many processes prepare the minibatches.
   1. Time one full pass of `train_dataloader` for `FashionMNIST` constructed with `batch_size` of 1, 16, 64, 256, and 1024, and plot the throughput in images per second against `batch_size`. Explain the shape of the curve in terms of per-batch overhead and per-image work.
   2. `get_dataloader` passes the `num_workers` attribute, which `DataModule` sets to 4 by default, to `torch.utils.data.DataLoader`. For `data = FashionMNIST()`, time one pass of `data.train_dataloader()` with `data.num_workers` set to 0, 1, 2, 4, and 8, where 0 loads in the main process. State under what conditions more workers stop helping.
5. code **Upsampling and linear models.** ⊠ Treat the resizing from 28×28 to 32×32 as a linear map $\mathbf{x}' = \mathbf{A}\mathbf{x}$ from the 784 original pixels to the 1024 resized ones, where the columns of $\mathbf{A} \in \mathbb{R}^{1024 \times 784}$ are linearly independent.
   1. Show that softmax regression on $\mathbf{x}'$ can represent exactly the classifiers that softmax regression on $\mathbf{x}$ can.
   2. After Section 3.3, train `SoftmaxRegressionScratch` on data from `FashionMNIST()` and

from `FashionMNIST(resize=(32, 32))`, with `num_inputs` set to the number of pixels per image. Compare the validation accuracies and explain any difference in light of the previous part.

3.3 Softmax Regression from Scratch

In Section 2.3 we wrote linear regression as a handful of plain functions, with tensors and automatic differentiation, so that no part of training stayed hidden, and in Section 2.4 we moved the training loop into a `Trainer` so that we would never write it again. We now write softmax regression the same way. What changes when the model predicts a class instead of a number? Rather little, as it turns out. Softmax regression differs from linear regression in three places. The model computes one output per class instead of a single number, the softmax turns these outputs, the logits, into probabilities, and the loss is the cross-entropy instead of the squared error. Everything else carries over: the `Trainer` runs the same loop with the same minibatch SGD update, and the `FashionMNIST` class of Section 3.2 delivers the images.

One piece of software is new, and every classifier in the book will use it. A classifier is trained on its loss but judged by its accuracy, so we write a base class `Classifier` that reports both. We write it, write the three new pieces, and train the model on Fashion-MNIST. Writing the softmax from its formula also exposes a numerical trap, which we demonstrate here and leave to the next section to remove. Once the model is trained, the interesting part begins. We look at what it gets wrong, at which garments it takes for which, and, since the weights of a linear model on images are themselves images, at what it has learned.

3.3.1 The Softmax

3.3.1.1 Normalizing Each Row

We start with the softmax, the piece that turns the logits of the model into probabilities. For a minibatch of n examples the logits form a matrix $\mathbf{O} = \mathbf{X}\mathbf{W} + \mathbf{b}$ with one row per example and one column per class (Section 3.1.1.4). The softmax acts on each row separately,

$$\operatorname{softmax}(\mathbf{O})_{ij} = \frac{\exp(O_{ij})}{\sum_{k=1}^{q} \exp(O_{ik})}, \tag{3.18}$$

so entry (i, j) of the result is the probability that the model assigns to class j for example i. Computing it takes three steps: exponentiate every entry, sum each row, and divide each row by its sum. Only the second step needs care with shapes. A sum with `axis=1` collapses the columns, and `keepdims=True` keeps the collapsed axis with length 1, so the sums form a column of shape $(n, 1)$ that broadcasts against the $n \times q$ matrix in the division (Section 1.1.3.2).

```
def softmax(O):
    O_exp = d2l.exp(O)
    return O_exp / d2l.reduce_sum(O_exp, 1, keepdims=True)  # Broadcast
```

We check it on two rows of logits. The first holds the logits $(1.0, 2.2, 0.3)$ that Section 3.1 computed for an image scored as cat, chicken, and dog; the second holds three equal logits.

```
O = d2l.tensor([[1.0, 2.2, 0.3], [0.0, 0.0, 0.0]])
Y_hat = softmax(O)
Y_hat, d2l.reduce_sum(Y_hat, 1)
```

```
(tensor([[0.2076, 0.6893, 0.1031],
         [0.3333, 0.3333, 0.3333]]),
 tensor([1., 1.]))
```

The first row reproduces the probabilities $(0.21, 0.69, 0.10)$ computed by hand, and equal logits give every class the probability $1/3$. Both rows sum to 1, as probabilities must.

3.3.1.2 Overflow

Unfortunately, the formula fails for large logits. For the logits $(1000, 0, 0)$, $\exp(1000)$ exceeds the largest single-precision number and becomes infinite, in the numerator of the first entry and in the row sum alike, and ∞/∞ is NaN, although the correct answer is $(1, 0, 0)$ up to rounding. The remedy is the invariance of Section 3.1: the softmax ignores a constant added to all logits of a row, so we may subtract the largest logit of each row first. Every exponent is then at most 0, and nothing overflows. The framework's built-in softmax shifts the logits in the same way. We apply the direct formula, the shifted formula, and the built-in softmax to the same row:

```
O = d2l.tensor([[1000.0, 0.0, 0.0]])
O_max = O.amax(dim=1, keepdim=True)
softmax(O), softmax(O - O_max), torch.softmax(O, dim=1)
```

```
(tensor([[nan, 0., 0.]]), tensor([[1., 0., 0.]]), tensor([[1., 0., 0.]]))
```

The direct formula returns NaN in the first entry; the shifted formula and the built-in softmax return $(1, 0, 0)$. A NaN does not stay where it arises. The loss of the minibatch becomes NaN, then its gradient, and after the update the parameters, and the model is lost. Our model is in no danger, since its inputs are pixel values between 0 and 1 and its weights stay small, so we keep the direct formula, which shows the three steps plainly. Nor is the shift the whole answer. It saves the softmax but not the loss, which takes the logarithm of the result, and the logarithm of a probability that has underflowed to 0 is $-\infty$. Section 3.4 removes both problems at once, by computing the logarithms of the probabilities directly from the logits.

3.3.2 Loss and Accuracy

Training a classifier lowers its cross-entropy loss. The user of a classifier has little interest in the loss and asks a different question: how often is the decision right? Every classifier in this book therefore reports two numbers on validation data: the loss, because training lowers it, and the accuracy, because users ask for it. Both go into a base class `Classifier` that extends the `Module` class of Section 2.4, so that each model supplies only its network and its loss.

3.3.2.1 Two Uses of the Logits

A classifier for q classes computes a vector of logits $\mathbf{o} \in \mathbb{R}^q$ for each example and predicts the class with the largest logit. Since the softmax preserves the order of the logits, this is also the class with the largest probability. Let $y^{(i)} \in \{1, \dots, q\}$ be the class of example i.

The accuracy on n examples is the fraction of examples whose largest logit belongs to the true class,

$$\text{accuracy} = \frac{1}{n}\sum_{i=1}^{n} \mathbf{1}\Big(\operatorname*{argmax}_j o_j^{(i)} = y^{(i)}\Big), \tag{3.19}$$

where $\mathbf{1}(\cdot)$ is 1 if its argument is true and 0 otherwise: a right prediction counts 1, a wrong one 0, and the accuracy is their average. A classifier that labels 9,000 of the 10,000 Fashion-MNIST validation images correctly has accuracy 0.9; its error rate, one minus the accuracy, is 0.1.

We report the accuracy but cannot train on it, since it changes only in jumps as the parameters change (Section 3.1). Gradient descent needs the smooth cross-entropy loss instead. Luckily, both numbers come from the same vector of logits, as Figure 3.5 shows.

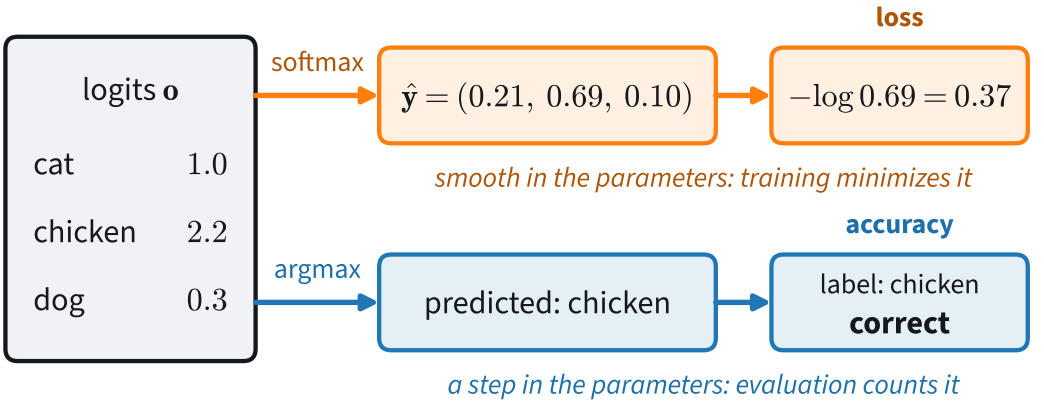


Figure 3.5: One logit vector, two uses. Training (top) turns the logits into probabilities with the softmax and reads off the cross-entropy loss of the true class, which changes smoothly with the parameters. Evaluation (bottom) takes the class with the largest logit and compares it with the label, which counts toward the accuracy and changes in steps. The logits $(1.0, 2.2, 0.3)$ score the classes cat, chicken, and dog for an image of a chicken.

It is tempting to assume that the two numbers at least move together. They need not. A model that becomes confidently wrong on a few validation examples pays a large loss on each of them, so its validation loss can rise while its accuracy still improves. This is why we report both.

3.3.2.2 The `Classifier` Class

`Classifier` has a single job: on each validation minibatch, compute the loss and the accuracy. To this end its `validation_step` replaces that of `Module`, which returns only the loss, by one that returns a dictionary with both metrics. The `Trainer` of Section 2.4 does the rest. It averages each metric that a step returns over the validation set, weighting every minibatch by its number of examples, so that the reported loss and accuracy are exact means over the validation set. After each epoch it appends the two means to `trainer.history['val_loss']` and `trainer.history['val_acc']`.

```
class Classifier(d2l.Module):
    """The base class of classification models."""
    def validation_step(self, batch):
        Y_hat = self(*batch[:-1])
        return {'loss': self.loss(Y_hat, batch[-1]),
                'acc': self.accuracy(Y_hat, batch[-1])}
```

This is all there is to the validation step. What the class does not contain is a network or a loss. These belong to its subclasses, and the rest of this section writes both for softmax regression.

3.3.2.3 Computing Accuracy

The method `accuracy` evaluates Equation 3.19 on a minibatch. `Y_hat` holds the outputs of the model, one row per example (more leading axes are flattened first); logits and probabilities have their largest entry in the same place. `argmax` along the class axis gives the predicted classes. We cast them to the data type of the labels `Y`, a formality that is needed because some frameworks compare only values of the same type. The comparison yields 1 for each correct prediction and 0 for each wrong one, and the mean of these values is the accuracy. With `averaged=False` the method returns the vector of 0s and 1s itself. This is the form to ask for when we want to look at the examples that the model gets wrong.

```
@d2l.add_to_class(Classifier)
def accuracy(self, Y_hat, Y, averaged=True):
    """Compute the fraction of correct predictions."""
    Y_hat = d2l.reshape(Y_hat, (-1, Y_hat.shape[-1]))
    preds = d2l.astype(d2l.argmax(Y_hat, axis=1), Y.dtype)
    compare = d2l.astype(preds == d2l.reshape(Y, (-1,)), d2l.float32)
    return d2l.reduce_mean(compare) if averaged else compare
```

3.3.2.4 The Default Optimizer

Unless a model asks for something else, it trains with minibatch stochastic gradient descent at its learning rate `self.lr`, as the linear regression of Chapter 2 did. The `Trainer` obtains the optimizer once, at the start of training, by calling the model's method `configure_optimizers`. Nothing in this default is specific to classification, so we install it on `Module` rather than on `Classifier`, and regression models inherit it as well. Later chapters override it to train with other optimizers.

```
@d2l.add_to_class(d2l.Module)
def configure_optimizers(self):
    return torch.optim.SGD(self.parameters(), lr=self.lr)
```

3.3.3 The Model and the Loss

3.3.3.1 The Parameters

Each 28×28 image becomes a vector of $d = 784$ pixel values, and there are $q = 10$ classes. The model therefore has a weight matrix $\mathbf{W} \in \mathbb{R}^{784 \times 10}$, one column per class, and a bias

vector $\mathbf{b} \in \mathbb{R}^{10}$, together 7,850 parameters. As in Section 2.3, we draw the weights from a normal distribution with mean 0 and standard deviation 0.01 and set the biases to 0. Such small weights give all classes nearly equal logits, so the model starts with predictions close to uniform, $\hat{y}_j \approx 0.1$, and a loss close to $\log 10 \approx 2.30$. The number is worth remembering: a first minibatch that reports a loss far from 2.30 usually points to a bug.

We register $\mathbf{W}$ and $\mathbf{b}$ as parameters of the module with the framework's parameter type, as in Section 2.4. The default optimizer that `Classifier` inherits from `Module`, minibatch SGD at the learning rate `lr`, finds registered parameters by itself, so this model needs no `configure_optimizers` of its own.

`nn.Parameter` registers each tensor with the module, so `model.parameters()` lists `W` and `b`.

```
class SoftmaxRegressionScratch(Classifier):
    """Softmax regression implemented from scratch."""
    def __init__(self, num_inputs, num_outputs, lr, std=0.01):
        super().__init__()
        self.save_hyperparameters()
        self.W = nn.Parameter(d2l.normal(0, std, (num_inputs, num_outputs)))
        self.b = nn.Parameter(d2l.zeros(num_outputs))
```

3.3.3.2 The Forward Pass

A minibatch of n images arrives as a four-dimensional tensor (Section 3.2). The method `forward` reshapes it into the $n \times 784$ matrix $\mathbf{X}$, one flattened image per row, computes the logits $\mathbf{O} = \mathbf{X}\mathbf{W} + \mathbf{b}$, with the bias added to every row by broadcasting, and returns their softmax. Flattening discards which pixels are neighbors. The model cannot use that information; the convolutional networks of Chapter 6 do.

The minibatch has shape $(n, 1, 28, 28)$, channels first.

```
@d2l.add_to_class(SoftmaxRegressionScratch)
def forward(self, X):
    X = d2l.reshape(X, (-1, self.num_inputs))  # One flattened image per row
    return softmax(d2l.matmul(X, self.W) + self.b)
```

3.3.3.3 The Cross-Entropy Loss

The loss of one example is the negative logarithm of the probability that the model assigns to its true class, $l = -\log \hat{y}_y$ (Section 3.1). For a minibatch we need, from each row of $\hat{\mathbf{Y}}$, the entry in the column of that example's label. The indexing of Section 1.1 does this without a loop: the row positions together with the vector of labels pick one entry per row. In code the labels are class indices counted from 0. Here are two examples with three classes and labels 0 and 2:

```
y = d2l.tensor([0, 2])
y_hat = d2l.tensor([[0.1, 0.3, 0.6], [0.3, 0.2, 0.5]])
y_hat[[0, 1], y]
```

```
tensor([0.1000, 0.5000])
```

The first example gives its true class the probability 0.1, the second 0.5. The function `cross_entropy` averages the negative logarithms of these probabilities over the minibatch. Its one line that is not part of the definition raises each probability to at least 10^{-12} before the logarithm, a crude guard that turns the infinite loss of a probability that has underflowed to 0 into 27.6 and, as the comment warns, into a zero gradient; the next section replaces it by a loss that never forms the probability.

```
def cross_entropy(y_hat, y):
    # Tiny clip to keep log finite when softmax outputs underflow to 0.
    p = y_hat[list(range(len(y_hat))), y].clamp(min=1e-12)
    return -d2l.reduce_mean(d2l.log(p))

cross_entropy(y_hat, y)
```

```
tensor(1.4979)
```

The loss is the mean of $-\log 0.1 \approx 2.30$ and $-\log 0.5 \approx 0.69$, about 1.50. It becomes the loss of our model:

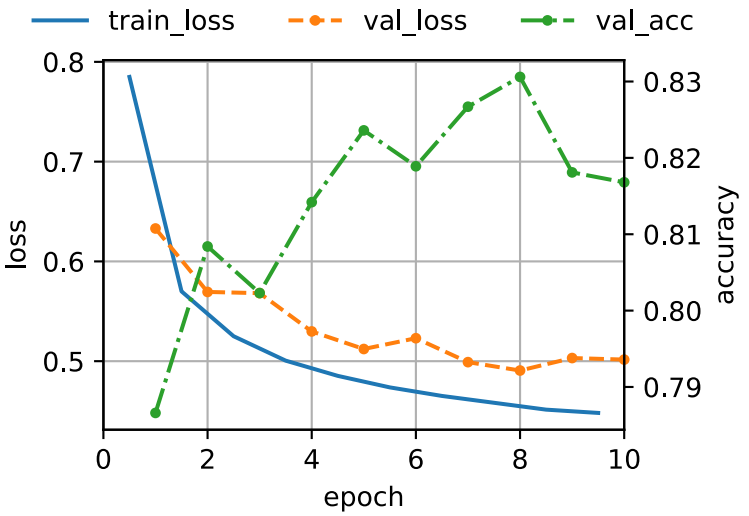
```
@d2l.add_to_class(SoftmaxRegressionScratch)
def loss(self, y_hat, y):
    return cross_entropy(y_hat, y)
```

3.3.4 Training and Evaluation

3.3.4.1 The Training Run

With the parameters, the forward pass, and the loss in place, the `Trainer` can fit the model to Fashion-MNIST. We train for 10 epochs with minibatches of 256 images at learning rate 0.1, which makes 235 steps per epoch, and the `Trainer` reports the loss and accuracy on the 10,000 validation images after every epoch. None of the three numbers we just chose is learned, yet each influences the result. Choosing them by their validation accuracy is a form of model selection, and Chapter 3.6 asks what such choices do to the trustworthiness of the score.

```
data = d2l.FashionMNIST(batch_size=256)
model = SoftmaxRegressionScratch(num_inputs=784, num_outputs=10, lr=0.1)
trainer = d2l.Trainer(max_epochs=10)
trainer.fit(model, data)
```



The training loss falls quickly during the first epoch and slowly after it. The validation accuracy is between 75% and 80% after one epoch and about 83% after ten. It does not rise steadily: from one epoch to the next it can drop by a point or two, occasionally more, since minibatch SGD keeps moving the parameters. The training loss ends near 0.45 and the validation loss only a few hundredths above it.

Validation metrics are also needed outside `fit`, for instance to evaluate a trained model once on a test set. We add the method `evaluate` to the trainer. It returns the means of the metrics of `model` over `data.val_dataloader()`, without drawing a single training minibatch, by calling the same `validate` that `fit` calls after every epoch (Section 2.4). With `train=True` it reads the training data instead, still without changing the parameters.

```
@d2l.add_to_class(d2l.Trainer)
def evaluate(self, model, data, train=False):
    """Metrics of `model` on the validation data of `data` (on its training
    data if `train`), averaged over examples."""
    self.train_dataloader = None
    self.val_dataloader = (data.train_dataloader() if train
                           else data.val_dataloader())
    self.prepare_model(model)
    return self.validate()
```

Here the metrics are means over all 10,000 validation images. After training they equal the last point of each validation curve, which `trainer.history['val_loss'][-1]` and `trainer.history['val_acc'][-1]` hold.

```
trainer.evaluate(model, data)
```

```
{'loss': 0.501600732421875, 'acc': 0.8168}
```

The model labels about 83% of the validation images correctly, and the number should not be quoted more precisely than that. Repeated runs land anywhere between 82% and 84%, since the random initialization and the order of the minibatches move the result by about

a point (Section 1.1 explains why we report such ranges).

How does this compare with the training images? The training curve above averages the loss over each epoch while the parameters still change, so we evaluate the trained model on all 60,000 training images.

```
trainer.evaluate(model, data, train=True)
```

```
{'loss': 0.46417646484375, 'acc': 0.8347333333333333}
```

The training accuracy is about 84%, one or two points above the validation accuracy: the model does almost as well on images it has not seen as on the images it was fitted to. A gap this small is no accident. A linear model cannot fit arbitrary labels of 60,000 images, so it can memorize little of its training data, and Chapter 3.6 says how large the gap could have been.

3.3.4.2 Misclassified Images

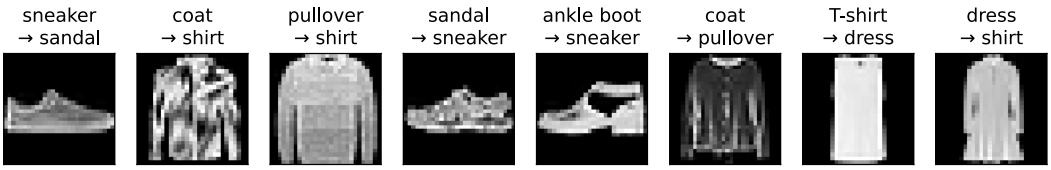
The accuracy tells us how often the model errs, not on which images. We take the first minibatch of 256 validation images, predict the class of each, and count the mistakes.

```
X, y = next(iter(data.val_dataloader()))
preds = d2l.astype(d2l.argmax(model(X), axis=1), y.dtype)
wrong = preds != y
print(f'{int(d2l.reduce_sum(d2l.astype(wrong, d2l.int32)))} of {len(y)} wrong')
```

```
44 of 256 wrong
```

We show the first eight misclassified images. Each title gives the true class above the predicted one.

```
labels = [f'{true}\n→ {pred}' for true, pred in zip(
    data.text_labels(y[wrong]), data.text_labels(preds[wrong]))]
data.visualize([X[wrong], y[wrong]], labels=labels)
```



Between 30 and 45 of the 256 images are misclassified, in line with an error rate of about 17%. The mistakes are not wild ones. Most of them take one top for another or one shoe for another. In fairness to the model, some of these images are hard to label at 28×28 pixels even for a person.

3.3.4.3 The Confusion Matrix

Eight images are anecdotes. A *confusion matrix* **N** counts all the mistakes. For q classes it has $q \times q$ entries. Let $\hat{c}^{(m)} = \text{argmax}_j \, o_j^{(m)}$ be the predicted class of example m, the class

with the largest logit; we keep $\hat{\mathbf{y}}$ for the predicted probabilities. Then

$$N_{ij} = \sum_{m=1}^{n} \mathbf{1}(\hat{c}^{(m)} = i)\, \mathbf{1}(y^{(m)} = j) \tag{3.20}$$

counts the examples of class j predicted as class i: rows are predicted classes and columns true classes. The diagonal holds the correct predictions, so the accuracy is the sum of the diagonal divided by n, and each off-diagonal entry counts one kind of mistake. Libraries do not agree on the orientation; scikit-learn's `confusion_matrix`, for one, puts the true classes in the rows. We put them in the columns for a reason that Section 3.7 will use. Once each column is divided by its sum, the matrix acts on a vector of class proportions and returns the frequencies of the predictions, $\mathbf{C}\mathbf{p} = \mu$, an equation that can be solved for $\mathbf{p}$.

How do we count 100 kinds of pairs without a loop over examples? One-hot vectors (Section 3.1) do it. Let the rows of $\mathbf{P} \in \{0, 1\}^{n \times q}$ be the one-hot encodings of the predicted classes of a minibatch and the rows of $\mathbf{Y} \in \{0, 1\}^{n \times q}$ those of the labels. Then

$$(\mathbf{P}^\top \mathbf{Y})_{ij} = \sum_{m=1}^{n} P_{mi} Y_{mj},$$

and a term is 1 exactly when example m is predicted as i and labeled j, as in Equation 3.20. One matrix product thus counts all pairs of a minibatch, and summing the products over the validation minibatches gives $\mathbf{N}$. The function `one_hot` builds the one-hot rows by comparing a column of class indices with the row $(0, 1, \ldots, q-1)$, which broadcasting expands to an $n \times q$ matrix of comparisons.

```
def one_hot(y, q):
    """Rows of the q x q identity matrix, one per class index in y."""
    return d2l.astype(d2l.reshape(y, (-1, 1)) == d2l.arange(q, dtype=y.dtype),
                      d2l.float32)

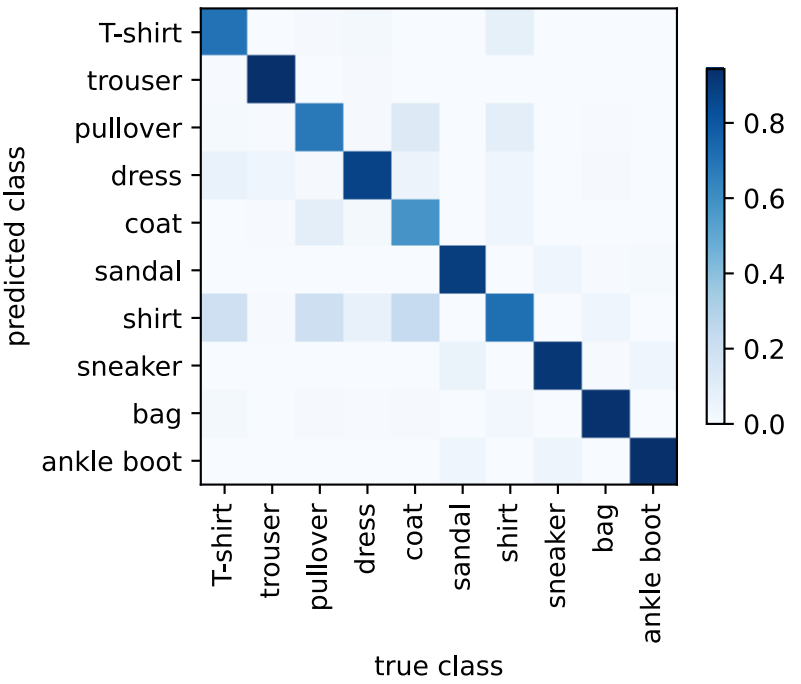
N = d2l.zeros((10, 10))
for X, y in data.val_dataloader():
    P = one_hot(d2l.argmax(model(X), axis=1), 10)
    N += d2l.matmul(d2l.transpose(P), one_hot(y, 10))
int(d2l.reduce_sum(N))
```

```
10000
```

The counts add up to the 10,000 validation images, so no pair was lost. Counts depend on how many examples each class happens to have. To compare classes, we divide each column of $\mathbf{N}$ by its sum. This gives the matrix $\mathbf{C}$ with $C_{ij} = N_{ij} / \sum_{i'} N_{i'j}$, the fraction of the examples of class j that are predicted as class i, an estimate of $P(\hat{c} = i \mid y = j)$. Column j shows where the examples of class j end up, and the diagonal holds, for each class, the fraction of its examples that the model recognizes. The display utility `d2l.show_heatmaps` draws $\mathbf{C}$ as a grid of colored cells.

```
C = N / d2l.reduce_sum(N, 0, keepdims=True)  # Each column sums to 1
names = data.text_labels(range(10))
```

```
d2l.show_heatmaps(d2l.reshape(C, (1, 1, 10, 10)), xlabel='true class',
                  ylabel='predicted class', figsize=(4, 4), cmap='Blues',
                  xticklabels=names, yticklabels=names)
```



The largest entries off the diagonal name the most frequent mistakes:

```
mistakes = d2l.numpy(N) * (1 - np.eye(10))  # Zero the diagonal
for k in np.argsort(mistakes, axis=None)[::-1][:5]:
    i, j = divmod(int(k), 10)
    print(f'{names[j]} taken for {names[i]}: {int(mistakes[i, j])}')
```

```
coat taken for shirt: 234
pullover taken for shirt: 202
T-shirt taken for shirt: 197
coat taken for pullover: 123
shirt taken for pullover: 95
```

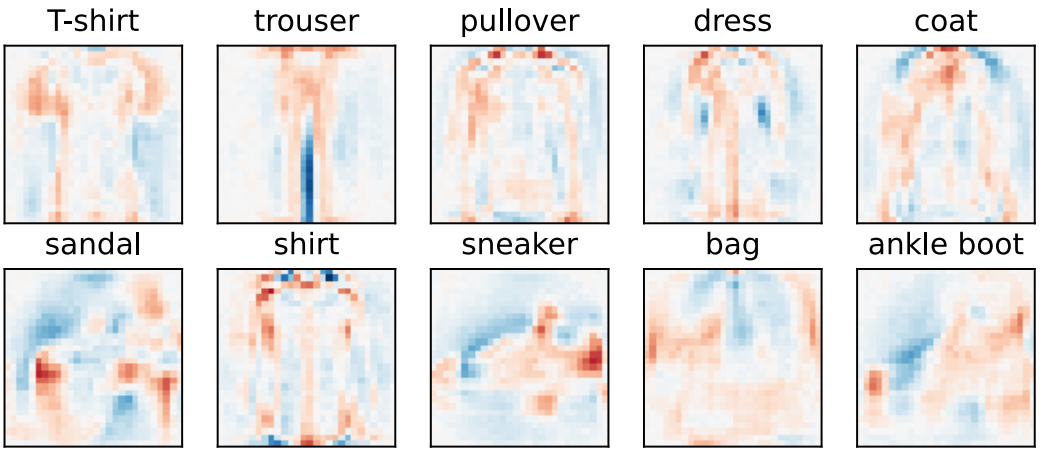
The diagonal shows how often each class is recognized. Trousers, bags, and the three kinds of shoes are recognized about 90% of the time or more, and shirts only about half the time, between 40% and 60% depending on the run. The errors are far from evenly spread. They form the two groups that Section 3.2 led us to expect. The four tops (T-shirts, pullovers, coats, and shirts) trade most of their errors among themselves: in every run, the five largest entries off the diagonal are pairs of tops, and the largest of them, a pullover taken for a coat or for a shirt, or a coat taken for a pullover, holds between a sixth and a third of its class. Sandals, sneakers, and ankle boots form a second, smaller group, whose errors almost all stay among the three. The four tops have similar outlines and differ in details such as collars, buttons, and the length of the sleeves, which change few pixels. A shirt resembles each of

the other three in some of these details, so it collects errors from all of them.

3.3.4.4 The Learned Templates

Why these mistakes and not others? The weights tell. The score of class j is the inner product of the flattened image with column j of $\mathbf{W}$, plus a bias. Reshaped to 28×28 pixels, the column is itself an image, a *template* for class j: a bright pixel where the template is red raises the score of the class, and a bright pixel where it is blue lowers it. We draw the ten templates on one color scale, centered at 0.

```
W = d2l.numpy(model.W)
templates = W.T.reshape(10, 28, 28)  # Column j of W as an image
v = np.abs(templates).max()
d2l.show_images(templates, 2, 5, titles=names, cmap='RdBu_r', vmin=-v, vmax=v);
```



Each template is a blurred garment, and the ten of them are the whole model: to classify an image is to ask which template it overlaps most. Templates of the same form can also be counted instead of trained, from how often each pixel is inked in each class; that is the naive Bayes classifier of Section 36.11. The templates of trousers, bags, and shoes have outlines that no other class shares; the trouser template is blue between the legs, where a dress is bright and a trouser is dark. The four tops share most of their outline, and their templates differ mainly at the neckline and the sleeves: a few pixels, and the same few in which the confusion matrix says they are confused.

More training will not remove these errors. Every class score is a weighted sum of pixels, so the boundaries between the predicted classes are hyperplanes in the space of images, whatever the weights, and a collar is too few pixels to tip such a sum reliably. Models that first compute features from groups of pixels, such as the multilayer perceptrons of Chapter 4, separate the classes better. Before we go there, one piece of this implementation still needs repair: a softmax that returns `NaN`, and a loss that can only hide it.

3.3.5 Summary

Softmax regression from scratch adds three components to the training code of linear regression: a weight matrix $\mathbf{W} \in \mathbb{R}^{784 \times 10}$ and a bias $\mathbf{b} \in \mathbb{R}^{10}$ registered as parameters, a

forward pass that flattens the images and applies the softmax to each row of $\mathbf{O} = \mathbf{X}\mathbf{W} + \mathbf{b}$, and the cross-entropy loss, which picks the probability of the true class in each row by indexing. The base class `Classifier` reports, on every validation minibatch, the loss that training lowers and the accuracy that users ask for, which changes in steps and cannot be trained on. Ten epochs of minibatch SGD reach about 83% validation accuracy and about 84% on the training images. The confusion matrix, a sum of products of one-hot matrices, shows that most errors trade one top for another, and the learned weights, viewed as ten templates, show why: the tops differ in few pixels. The softmax written from its formula returns `NaN` when a large logit overflows the exponential, and the floor in the loss only hides the matching failure of the logarithm; the next section computes the loss from the logits instead.

3.3.6 Exercises

1. code **Underflow.** The `softmax` of this section returns `NaN` on the logits $(1000, 0, 0)$, and subtracting the largest logit of the row fixes it.
    1. Show that `softmax` also returns `NaN` on the row $(-1000, -1000, -1000)$, and identify the operation that produces the `NaN` in this case. Check that the shifted version returns $(1/3, 1/3, 1/3)$.
    2. Take the logits $(0, -200, 0)$ and the label `y = 1`, the class of the logit -200. Compute the shifted softmax and `cross_entropy` of the result. Then compute the gradient of this loss with respect to the logits by automatic differentiation, and compare it with the gradient $\hat{\mathbf{y}} - \mathbf{y}$ derived in Section 3.1. Explain the difference, and what it means for training on an example that the model gets badly wrong.
2. code **Cross-entropy from one-hot labels.** The `cross_entropy` of this section selects the correct-class probability by indexing. Implement an alternative that follows the definition $-\sum_j y_j \log \hat{y}_j$ literally, by converting `y` to a one-hot matrix and taking row-wise inner products.
    1. Use it as the `loss` of `SoftmaxRegressionScratch` and confirm that training is unchanged.
    2. Time both versions on random probability matrices with 256 rows and $q \in \{10, 10^3, 10^5\}$ columns, and state how the memory that each version allocates grows with q.
    3. Describe a setting in which the inner-product form applies but the indexing form does not.
3. **Large vocabularies.** Suppose that softmax regression predicts the next word from a feature vector of dimension $d = 1024$. The vocabulary has $V = 10^6$ words, and each minibatch holds $n = 256$ examples. List the operations of `forward` and `cross_entropy` whose cost grows with V. Compute the multiply-adds of the matrix product, the float32 memory of `W`, and the float32 memory of the $n \times V$ matrix of probabilities, and compare each number with the Fashion-MNIST model of this section.
4. code **Hyperparameter sweep.** Retrain `SoftmaxRegressionScratch` with `d2l.Trainer(max_epochs=10)`, varying one hyperparameter at a time, and read the final losses from `trainer.history`.
    1. Plot the final validation loss against the learning rate over $\{0.01, 0.03, 0.1, 0.3, 1\}$.
    2. Plot the final training and validation losses against the minibatch size over $\{8, 32, 256, 2048\}$ at learning rate 0.1.

3. Compute the number of parameter updates that ten epochs over the 60,000 training images take at each minibatch size. Use these counts to explain the trend in the previous plot, and predict how the comparison changes when every run takes the same number of updates.

5. code **Templates and class means.** The templates are what minibatch SGD made of the pixels it saw.
   1. Train `SoftmaxRegressionScratch` again with `std=0`, so that every weight starts at 0 and every class starts with the same logit. Explain why training works all the same, and compare the accuracy and the templates with those of the text.
   2. Show that after one SGD step from zero weights, column j of $\mathbf{W}$ is proportional to $\sum_i \left(y_j^{(i)} - \frac{1}{10}\right) \mathbf{x}^{(i)}$ over the first minibatch: the sum of the images of class j minus a tenth of the sum of all images.
   3. Compute the mean image of each class over the training set and compare it with the trained template of that class, by eye and by the correlation of their pixels. Where do they agree, and where does a template depart from the mean? Explain why a template that merely copied the class mean would confuse the tops even more.

3.4 Concise Implementation

In Section 3.3 we wrote every part of softmax regression by hand. In practice nobody does. We use the building blocks of a deep learning framework instead, as Section 2.5 did for linear regression: a fully connected layer that holds the weights and the bias, and a cross-entropy loss. If saving lines of code were all there is to it, this section could be very short. The framework's loss, however, differs from ours in one respect that matters: it takes the logits rather than the probabilities. Recall that the softmax we wrote by hand returns `NaN` once a logit grows large, and that the floor under its probabilities kept the loss finite only by cutting off the gradient. How can the loss be computed so that it stays finite, and keeps its gradient, for every value of the logits? The answer, it turns out, is never to form the probabilities at all.

3.4.1 The Model

As in Section 2.5, a fully connected layer of the framework holds the weights $\mathbf{W}$ and the bias $\mathbf{b}$ and computes $\mathbf{O} = \mathbf{X}\mathbf{W} + \mathbf{b}$. It has 10 outputs, one per class. The model flattens each image into a vector of 784 pixel values before the layer, as the scratch implementation did, and its `forward` returns the logits. Note that we set the initial values ourselves: the weights start from a normal distribution with mean 0 and standard deviation 0.01 and the biases at 0, again as in the scratch implementation. Each framework's layer would otherwise draw its weights from a default distribution of its own, and the four tabs would train different models.

As in Section 2.5, the lazy layer learns its input size at the first forward pass, and the `Trainer` calls `weight_init` after it.

```
class SoftmaxRegression(d2l.Classifier):
    """The softmax regression model."""
```

```
    def __init__(self, num_outputs, lr):
        super().__init__()
        self.save_hyperparameters()
        self.net = nn.Sequential(nn.Flatten(),
                                 nn.LazyLinear(num_outputs))

    def weight_init(self, module):
        # The Trainer calls this after the lazy layer has learned its shape
        if isinstance(module, nn.Linear):
            nn.init.normal_(module.weight, std=0.01)
            nn.init.zeros_(module.bias)

    def forward(self, X):
        return self.net(X)
```

3.4.2 A Stable Cross-Entropy Loss

3.4.2.1 Overflow and Underflow

The scratch implementation computes the probabilities first and takes the logarithm afterwards, and each step can fail in floating point. Single precision stores numbers up to about 3.4×10^{38}, so $\exp(o)$ overflows to infinity once o exceeds about 88, which is what turned the softmax of $(1000, 0, 0)$ into `NaN` in Section 3.3. This failure is the easy one. A common shift of all logits leaves the softmax unchanged, so subtracting the largest logit, $o_{\max} = \max_k o_k$, from each of them removes it, as we saw there. Unfortunately, the shift does nothing for the lower end. For the logits $(0, -200, 0)$ the probability of the second class is about $e^{-200}/2$, smaller than any positive single-precision number, and so it is stored as 0; this happens to every logit that lies more than about 104 below the largest one. The logarithm of this 0 is $-\infty$, and backpropagation through it produces `NaN`. The floor of the scratch loss hides the infinity and discards the gradient with it. Both failures have the same cause: we formed a probability that the loss never needed.

3.4.2.2 The Log-Sum-Exp Form

What does the loss need? Only the logarithm of one probability, and the logarithm of a quotient is a difference. Taking the logarithm of the softmax on paper, before anything is computed, removes the division:

$$\log \hat{y}_j = o_j - \log \sum_k \exp(o_k).$$

This is the key step. For an example of true class y, the loss $l = -\log \hat{y}_y$ becomes a function of the logits alone, with no probability in between. As it stands the sum can still overflow, so we pull the factor $\exp(o_{\max})$ out of it:

$$l(y, \mathbf{o}) = \log \sum_k \exp(o_k) - o_y = o_{\max} + \log \sum_k \exp(o_k - o_{\max}) - o_y. \tag{3.21}$$

The two sides are equal, but only the right side should be handed to a computer. All its exponents are at most 0, and one of them is equal to 0, so the sum lies between 1 and q and

its logarithm between 0 and $\log q$. Every term is an ordinary number, whatever the logits. For $(0, -200, 0)$ with the second class as the true one, the loss is $\log 2 + 200 \approx 200.7$: large, as it should be for so wrong a prediction, but finite. The first term, $\log \sum_k \exp(o_k)$, is the *log-sum-exp* of the logits that Section 3.1.2.2 differentiated.

3.4.2.3 The Framework Loss

The cross-entropy losses of the frameworks compute Equation 3.21 when they are given logits. They never form the probabilities, so neither the exponential of a large number nor the logarithm of 0 occurs. Their gradient is the gradient of this expression, $\partial l / \partial o_j = \text{softmax}(\mathbf{o})_j - y_j$ (Section 3.1.2.2), with no floor to cut it off.

We install the framework's loss as the method `loss` of `Classifier`, so that every classification model in the book inherits it. The method takes logits and integer labels. It first flattens all leading axes of both, since later models predict one class for each position of a sequence, and returns the mean loss, or with `averaged=False` the loss of every example.

In PyTorch the loss is `F.cross_entropy`.

```
@d2l.add_to_class(d2l.Classifier)
def loss(self, Y_hat, Y, averaged=True):
    Y_hat = d2l.reshape(Y_hat, (-1, Y_hat.shape[-1]))
    Y = d2l.reshape(Y, (-1,))
    return F.cross_entropy(
        Y_hat, Y, reduction='mean' if averaged else 'none')
```

The loss takes logits, not probabilities. A model whose `forward` ended with a softmax would hand it probabilities, and the loss would apply a second softmax to them. No error message appears, and the model still trains, but its loss can no longer come near 0. This is why the `forward` of `SoftmaxRegression` returns the logits.

We put the loss to the test on the logits $(1000, 0, 0)$ that broke the softmax of the previous section, once with label 0 and once with label 1. By Equation 3.21 the losses are $\log(1 + 2e^{-1000}) \approx 0$ and $1000 + \log(1 + 2e^{-1000}) \approx 1000$.

```
O = d2l.tensor([[1000.0, 0.0, 0.0], [1000.0, 0.0, 0.0]])
y = d2l.tensor([0, 1])
SoftmaxRegression(num_outputs=3, lr=0.1).loss(O, y, averaged=False)
```

```
tensor([  -0., 1000.])
```

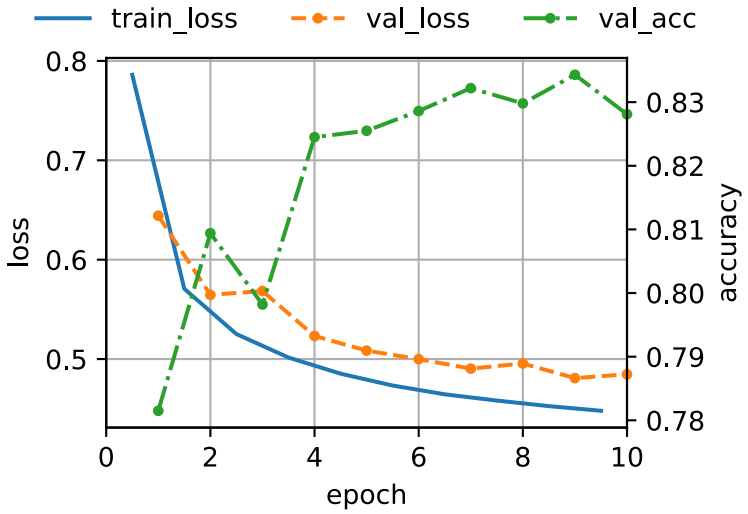
Both losses are finite and exact: the first example is classified correctly with certainty, and the second is wrong by a margin of 1000.

The first loss prints as `-0.`, a zero with its sign bit set, which equals 0.

3.4.3 Training

We train with the same data, minibatch size, learning rate, and number of epochs as in Section 3.3.

```
data = d2l.FashionMNIST(batch_size=256)
model = SoftmaxRegression(num_outputs=10, lr=0.1)
trainer = d2l.Trainer(max_epochs=10)
trainer.fit(model, data)
```



After ten epochs the validation accuracy is about 83%, as for the implementation from scratch, and the loss curves look alike. This is as it should be. The framework's components change how the model is computed, not the model. We now have a classifier, a loss that cannot blow up, and a number, 83%. The rest of the chapter asks what the number is worth, beginning with the probabilities behind it: can they be taken at face value?

3.4.4 Summary

Softmax regression with framework components is a fully connected layer on flattened images whose forward pass returns logits, trained with the framework's cross-entropy loss. The loss takes logits because computing it from probabilities fails in floating point, by overflow of the exponential or by the logarithm of a probability that has underflowed to 0. Written as $l = \log \sum_k \exp(o_k) - o_y$ and evaluated after subtracting the largest logit, the loss stays finite for any logits and keeps its gradient, $\text{softmax}(\mathbf{o}) - \mathbf{y}$. `Classifier.loss` installs this loss for every classifier in the book. Trained for ten epochs, the model reaches about 83% validation accuracy, like the version from scratch.

3.4.5 Exercises

1. code **Double softmax.** The `loss` of this section expects logits. Suppose that a model's `forward` mistakenly ends with a softmax. The loss then receives a probability vector $\mathbf{p}$ in place of the logits and evaluates $-\log \text{softmax}(\mathbf{p})_y$.
    1. Find the smallest value this loss can take over all probability vectors $\mathbf{p}$ and labels y, as a function of the number of classes q. Evaluate it for $q = 10$.
    2. Copy the from-scratch `softmax` of Section 3.3, which is not part of the `d2l` package. Subclass `SoftmaxRegression` so that `forward` applies this `softmax` to the logits. Train

it with `lr=0.1`, `d2l.FashionMNIST(batch_size=256)`, and `d2l.Trainer(max_epochs=10)`. Compare its training loss and validation accuracy with those of `SoftmaxRegression`, and relate the final training loss to the value from the previous step.

2. code **Fused loss on extreme logits.** The text passes the logits $\mathbf{o} = (1000, 0, 0)$ to the `loss` of this section with labels 0 and 1. Copy the from-scratch `softmax` of Section 3.3, which is not part of the `d2l` package.
   1. Compute the same two losses by the from-scratch route, `softmax` followed by `d2l.cross_entropy`. Explain each value it returns, and determine the largest finite loss this route can return.
   2. Verify on benign logits such as $\mathbf{o} = (2, 1, 0)$ that the two routes agree to floating-point precision.
3. code **Stable softplus.** For two classes with logits $(x, 0)$, the first term of the loss is $\log(1 + e^x)$, the *softplus* of x.
   1. Plot $\log(1+e^x)$ and $\max(x, 0)$ on $[-5, 5]$. Show that the first exceeds the second by at most $\log 2$, at $x = 0$, and explain in one sentence why log-sum-exp is called a smooth maximum.
   2. Show that $\log(1 + e^x) = \max(x, 0) + \log(1 + e^{-|x|})$.
   3. In single precision, evaluate the naive expression `log(1 + exp(x))`, the rewritten expression, and the rewritten expression with `log1p` in place of `log(1 + ...)`, for $x \in \{-100, -20, -5, 0, 5, 20, 100\}$. Compare each value with a double-precision reference, and report where each version overflows and where it loses all relative accuracy.
   4. Compute the derivative of the naive expression at $x = -20$ by automatic differentiation. Explain why it is accurate although the value of the expression is not, and what this implies for training and for reporting the loss of confidently classified examples.
4. code **The gradient survives.** In Section 3.3, exercise 1, the floored loss of the logits $(0, -200, 0)$ with label 1 had a zero gradient. Pass the same logits and label to the `loss` of this section and compute the gradient with respect to the logits by automatic differentiation. Compare it with $\hat{\mathbf{y}} - \mathbf{y}$ from Section 3.1, and explain why this gradient would move the logit of class 1 while the floored one did not.
5. **Number formats.** Half-precision formats store a number in 16 bits: FP16 has 5 exponent bits and 10 mantissa bits, BFLOAT16 has 8 and 7, and FP32 has 8 and 23. Determine for each format the largest logit o for which $\exp(o)$ does not overflow. Then explain why a network whose logits reach 20 can be trained in BFLOAT16 but not in FP16 unless its softmax is computed in FP32. Finally, consider the shifted sum of Equation 3.21 for $q = 10^5$ classes, each term at most 1, accumulated in FP16: what goes wrong once the running sum exceeds 2,048, and what does the 10-bit mantissa have to do with it?

3.5 Measuring Classifiers

The softmax regression of Section 3.3 and Section 3.4 labels about 83% of the Fashion-MNIST validation images correctly. Is a classifier with higher accuracy the better one? Only if every example matters equally and every mistake is equally bad, since accuracy counts every example once and treats every mistake alike. Three kinds of decision problem break

these assumptions, and they are the common ones: one class is much rarer than the others, some mistakes cost more than others, and the decision is made on the predicted probability itself rather than on the predicted class. Each calls for a measurement of its own. We study the first two on a screening test whose outcomes we can count by hand, and the third on the trained softmax regression, whose probabilities we have so far taken on trust.

3.5.1 The Limits of Accuracy

3.5.1.1 Class Imbalance

Consider screening for a disease that affects 1% of the population. A classifier that ignores its input and always predicts *healthy* is right for 99% of the people, an accuracy of 0.99, and finds no sick patient. Now take a screening test that detects 90% of the sick and wrongly flags 5% of the healthy. Among 100,000 people, 1,000 of them sick, it finds 900 and misses 100, and it flags 4,950 of the 99,000 healthy people. We count the four possible outcomes in a table whose rows are the predictions and whose columns are the true classes:

	sick	healthy
predicted sick	900	4,950
predicted healthy	100	94,050

Such a table of counts is a *confusion matrix*. With the sick as the *positive* class, the four cells are the *true positives* (TP, 900), *false positives* (FP, 4,950), *false negatives* (FN, 100), and *true negatives* (TN, 94,050). The test is right for the 94,950 people on the diagonal, an accuracy of 0.95. Accuracy ranks the useless classifier above the useful one. Every person counts once, so the 1,000 sick people make up only 1% of the accuracy, however well or badly a classifier treats them.

Two ratios focus on the sick. The *recall* is the fraction of the sick that the classifier finds, and the *precision* is the fraction of the flagged people who are sick:

$$\text{recall} = \frac{\text{TP}}{\text{TP} + \text{FN}}, \qquad \text{precision} = \frac{\text{TP}}{\text{TP} + \text{FP}}. \tag{3.22}$$

Recall reads the first column of the table, and precision reads its first row. For the screening test the recall is $900/1{,}000 = 0.9$ and the precision $900/5{,}850 \approx 0.15$.

Recall separates the two classifiers: 0 for the one that never flags anybody, 0.9 for the test. The precision of the always-healthy classifier is undefined, since it flags nobody. The precision of the test is a less pleasant surprise. It is the probability that a flagged person is sick, and it is only 0.15. This is the effect of Bayes' theorem that Section 1.6.5.4 showed for an HIV test: when a disease is rare, the false positives from the many healthy people outnumber the true positives, even for an accurate test. Is a test with high recall and low precision any good? For screening it is, since every flagged person gets a second, more specific test.

How much does the precision owe to the rarity of the disease? We keep the test, which finds 90% of the sick and flags 5% of the healthy, and vary the fraction of sick people in the population.

```
recall, false_alarms = 0.9, 0.05  # Properties of the test
for prevalence in [0.001, 0.01, 0.1, 0.5]:
    tp, fp = recall * prevalence, false_alarms * (1 - prevalence)
    accuracy = tp + (1 - false_alarms) * (1 - prevalence)
    print(f'{prevalence:5.1%} sick: accuracy {accuracy:.3f}, '
          f'recall {recall:.2f}, precision {tp / (tp + fp):.3f}')
```

```
 0.1% sick: accuracy 0.950, recall 0.90, precision 0.018
 1.0% sick: accuracy 0.950, recall 0.90, precision 0.154
10.0% sick: accuracy 0.945, recall 0.90, precision 0.667
50.0% sick: accuracy 0.925, recall 0.90, precision 0.947
```

The same test has a precision of 2% when one person in a thousand is sick and of 95% when half of them are, while its accuracy hardly moves. Recall and the rate of false alarms describe the test; precision describes the test in a population.

3.5.1.2 Asymmetric Costs

Missing a sick patient is worse than a false alarm. Suppose that a false negative costs 100 times as much as a false positive, $\kappa_{\text{FN}} = 100$ and $\kappa_{\text{FP}} = 1$. The cost of a classifier weights the cells of its confusion matrix by these costs. Per person, the always-healthy classifier costs $100 \cdot 1{,}000/100{,}000 = 1$ and the screening test $(100 \cdot 100 + 1 \cdot 4{,}950)/100{,}000 \approx 0.15$, almost seven times less. With costs, the useful classifier wins.

Costs also change the decision itself. Suppose a model estimates the probability p that a patient is sick. Predicting *healthy* costs κ_{FN} with probability p, and predicting *sick* costs κ_{FP} with probability $1 - p$, so the expected cost is smaller when we predict *sick* exactly if

$$p\,\kappa_{\text{FN}} > (1-p)\,\kappa_{\text{FP}}, \quad \text{that is,} \quad p > \frac{\kappa_{\text{FP}}}{\kappa_{\text{FP}} + \kappa_{\text{FN}}}. \tag{3.23}$$

With equal costs the threshold is $\frac{1}{2}$, and for two classes the rule picks the class with the larger probability, as accuracy assumes. With $\kappa_{\text{FN}} = 100$ and $\kappa_{\text{FP}} = 1$, the threshold drops to $1/101 \approx 0.01$: a patient with a 2% chance of being sick should be flagged. Lowering the threshold raises the recall and lowers the precision, since more people are flagged. When the costs are unknown, practitioners often report precision and recall together, or their harmonic mean, the *F1 score*, which is high only when both are.

For q classes, costs κ_{ij} for predicting i when the class is j weight the entries of the confusion matrix $\mathbf{N}$ of Equation 3.20 as in the binary case.

3.5.2 Calibration

3.5.2.1 Reliability Diagrams

The rule Equation 3.23 acts on the probability p itself, so its decisions are only as good as the probabilities. A classifier is *calibrated* if its probabilities match frequencies: of all patients to whom it assigns $p = 0.02$, 2% are sick. If only 0.2% of them are, the rule flags all of them, although at the threshold of 0.01 none of them should be flagged. Checking this on validation data is simple. We sort the patients into bins by p and compare the mean of p in each bin with the fraction of its patients who are sick.

With many classes there is one probability per class to check, and the usual shortcut is to check only the largest. The *confidence* of a prediction is the probability of the predicted class. We sort the examples into bins by their confidence, for instance $[0.5, 0.7)$, $[0.7, 0.9)$, and $[0.9, 1]$, and compute in each bin the mean confidence and the accuracy. A *reliability diagram* plots the accuracy of each bin against its mean confidence. A calibrated classifier lies on the diagonal; points below it mean that the classifier is more confident than it is right, or *overconfident*, a common failing of deep networks trained on cross-entropy. Note, however, that this check says nothing about the small probabilities: a rule that thresholds $p(\text{sick})$ near 0.01 needs the check on $p(\text{sick})$ itself.

We check the softmax regression of Section 3.4. We train it as there, with the progress plot switched off since it would repeat the plot of that section, and collect the logits and the labels of all validation images as NumPy arrays.

```
data = d2l.FashionMNIST(batch_size=256)
model = d2l.SoftmaxRegression(num_outputs=10, lr=0.1)
trainer = d2l.Trainer(max_epochs=10)
trainer.board.display = False
trainer.fit(model, data)
O, y = [], []
for X, Y in data.val_dataloader():
    O.append(d2l.numpy(model(X)))
    y.append(d2l.numpy(Y))
O, y = np.concatenate(O), np.concatenate(y)
print(f'validation accuracy {(O.argmax(axis=1) == y).mean():.3f}')
```

```
validation accuracy 0.834
```

The confidence of each prediction is the largest entry of the softmax of its logits. We compute the softmax in NumPy, shifted by the largest logit as in Section 3.3.

```
def softmax(O):
    """Row-wise softmax of a NumPy array of logits."""
    E = np.exp(O - O.max(axis=1, keepdims=True))
    return E / E.sum(axis=1, keepdims=True)

conf = softmax(O).max(axis=1)  # Confidence of each prediction
correct = (O.argmax(axis=1) == y).astype(float)
```

We sort the predictions into ten bins by their confidence, $[0, 0.1)$, $[0.1, 0.2)$, and so on up to $[0.9, 1]$, and compute in each bin b the number of predictions n_b, their mean confidence $\bar{c}_b$, and their accuracy a_b. The first bin stays empty, since with $q = 10$ classes the largest probability is at least $1/10$. The expected calibration error (ECE) is the mean gap between confidence and accuracy, weighted by the fraction of the n predictions that fall into each bin,

$$\text{ECE} = \sum_b \frac{n_b}{n} \left| a_b - \bar{c}_b \right| .$$

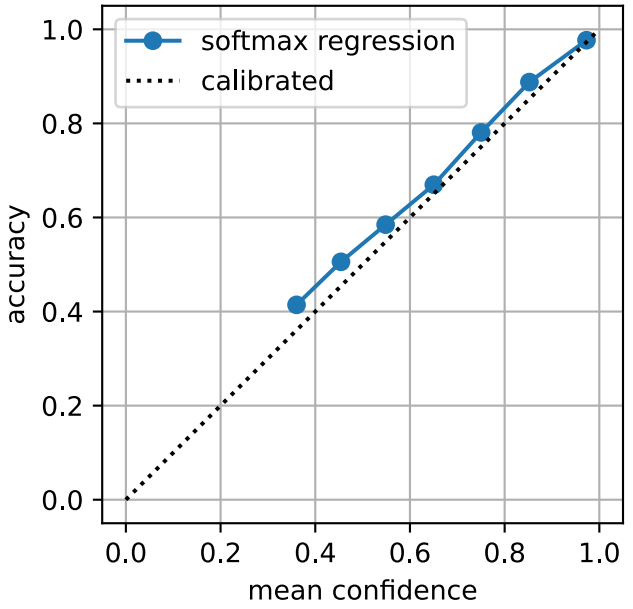
```
def reliability(conf, correct, num_bins=10):
    """Count, mean confidence, and accuracy per confidence bin; the ECE."""
    bins = np.minimum((conf * num_bins).astype(int), num_bins - 1)
    n = np.bincount(bins, minlength=num_bins)
    mean_conf = np.bincount(bins, conf, num_bins) / np.maximum(n, 1)
    acc = np.bincount(bins, correct, num_bins) / np.maximum(n, 1)
    return n, mean_conf, acc, np.sum(n * np.abs(acc - mean_conf)) / len(conf)

n, mean_conf, acc, ece = reliability(conf, correct)
for b in np.flatnonzero(n):
    print(f'confidence {b / 10:.1f} to {(b + 1) / 10:.1f}: {n[b]:4d} '
          f'predictions, mean {mean_conf[b]:.3f}, accuracy {acc[b]:.3f}')
print(f'ECE: {ece:.3f}')
```

```
confidence 0.1 to 0.2:    3 predictions, mean 0.183, accuracy 0.000
confidence 0.2 to 0.3:   51 predictions, mean 0.263, accuracy 0.294
confidence 0.3 to 0.4:  309 predictions, mean 0.360, accuracy 0.414
confidence 0.4 to 0.5:  609 predictions, mean 0.454, accuracy 0.506
confidence 0.5 to 0.6:  891 predictions, mean 0.549, accuracy 0.585
confidence 0.6 to 0.7:  893 predictions, mean 0.650, accuracy 0.670
confidence 0.7 to 0.8:  967 predictions, mean 0.750, accuracy 0.781
confidence 0.8 to 0.9: 1355 predictions, mean 0.852, accuracy 0.888
confidence 0.9 to 1.0: 4922 predictions, mean 0.973, accuracy 0.977
ECE: 0.020
```

About half of the predictions fall into the top bin, with a mean confidence near 0.97, and 97% to 98% of them are correct. The two lowest bins hold between a handful and about 70 predictions, too few to say anything about their accuracy. From confidence 0.3 up, every bin holds hundreds of predictions, and its accuracy lies near its mean confidence, in most bins slightly above it. The reliability diagram shows the bins with at least 100 predictions, whose accuracy has a standard error of at most 5 points.

```
used = n >= 100  # Fewer predictions give too noisy an accuracy
d2l.plot([mean_conf[used], [0, 1]], [acc[used], [0, 1]], 'mean confidence',
         'accuracy', legend=['softmax regression', 'calibrated'],
         fmts=('o-', 'k:'), figsize=(3.5, 3.5))
```



The curve stays close to the diagonal, and mostly just above it. Over all validation images the mean confidence is about 81% and the accuracy about 83%, and the expected calibration error stays below 0.03 in repeated runs. The model is slightly *under*confident, and its probabilities can be taken nearly at face value on images like its validation images. This is a property of the trained model, not of the softmax, and the same check applies unchanged to every classifier that follows. Accuracy, which uses only the predicted class, could not have told us any of it.

3.5.2.2 Temperature Scaling

When the check finds a gap, the standard remedy is as simple as the diagnosis. *Temperature scaling* divides all logits by one number $T > 0$, the temperature of Section 3.1, before the softmax, and fits T by minimizing the cross-entropy on held-out data (C. Guo et al. 2017). Dividing by T keeps the order of the logits, so every prediction and the accuracy stay the same. Only the confidences move, and with $T > 1$ they move down, the direction an overconfident network needs. Our model needs the opposite correction, if any. Its confidences are slightly too low, so the fitted temperature should come out below 1. We fit T on the first half of the validation images, by a grid search that minimizes the cross-entropy written in the stable form of Section 3.4, and measure the calibration error on the second half before and after scaling.

```
def nll(O, y, T):
    """Mean cross-entropy of softmax(O / T), computed from the logits."""
    Z = O / T
    m = Z.max(axis=1)
    lse = m + np.log(np.exp(Z - m[:, None]).sum(axis=1))
    return np.mean(lse - Z[np.arange(len(y)), y])

half = len(y) // 2
```

```
Ts = np.linspace(0.5, 2, 151)
T = Ts[np.argmin([nll(O[:half], y[:half], t) for t in Ts])]
ece = lambda Z: reliability(softmax(Z).max(axis=1), correct[half:])[3]
print(f'T = {T:.2f}; ECE on the second half: {ece(O[half:]):.3f} before, '
      f'{ece(O[half:] / T):.3f} after scaling')
```

```
T = 0.91; ECE on the second half: 0.016 before, 0.013 after scaling
```

The fitted temperature lies a little below 1, around 0.9, and scaling moves the calibration error of the second half by less than a hundredth, in some runs up rather than down. Softmax regression needed little repair, and at errors this small the binned estimate is as much noise as signal. For an overconfident network the same three lines return a T above 1 and a visibly straighter reliability diagram; the procedure does not care which way the gap points.

All of these numbers, from the accuracy to the calibration error, were measured on 10,000 images and would come out a little different on another 10,000. How different, and how far they can be trusted for images the model has never seen, is the subject of Chapter 3.6.

3.5.3 Summary

Accuracy counts every example once and every mistake alike. When one class is rare, it can rank a useless classifier above a useful one; recall and precision separate them, and precision depends on how common the class is. Costs of mistakes turn the predicted probabilities into a decision threshold, which is only as good as the probabilities are calibrated. A reliability diagram compares confidence with accuracy, and the expected calibration error summarizes it. Softmax regression on Fashion-MNIST is nearly calibrated, slightly underconfident; temperature scaling, which divides the logits by one fitted number, leaves its accuracy unchanged and changes its calibration little.

3.5.4 Exercises

1. **A small confusion matrix.** A classifier for three classes is evaluated on 300 validation examples, 100 of each class. With predictions in the rows and true classes in the columns, as in Equation 3.20, its confusion matrix is

$$\mathbf{N} = \begin{pmatrix} 90 & 20 & 0 \\ 10 & 70 & 10 \\ 0 & 10 & 90 \end{pmatrix}. \tag{3.24}$$

    1. Compute the accuracy, and the recall and the precision of each class.
    2. Divide each column of $\mathbf{N}$ by its sum, then each row by its sum. State what the diagonal of each result holds.
    3. Suppose the second class becomes ten times as common as each of the others while the classifier stays the same. Compute the new accuracy and the new precision of the second class, and say which of the numbers of the first part are properties of the classifier alone.

2. **Optimal decisions and rejection.** In a multiclass problem, let κ_{ij} be the cost of predicting class i when the true class is j, and let $p(j \mid x)$ be the probability of class j given the input x.

1. Write the expected cost of predicting i in terms of the costs and $p(j \mid x)$, and state the rule that selects the optimal i. Check that it reduces to Equation 3.23 for two classes.
2. Let a correct guess cost 0 and a wrong guess cost λ_s, and allow the classifier to *abstain* instead of guessing, at a fixed cost $\lambda_r < \lambda_s$. Derive the threshold on the top-class probability above which guessing beats abstaining.
3. In medical diagnosis, deferring to a specialist is cheap relative to a misdiagnosis, say $\lambda_r/\lambda_s = 0.05$. Compute the threshold and state what the classifier should do with a prediction at 0.9 confidence. Then describe a case in which reporting the full distribution $p(y \mid x)$ serves the physician better than either guessing or abstaining.

Adapted from Kevin Murphy, Probabilistic Machine Learning: An Introduction, Exercise 5.1.

3. **Loss at equal accuracy.** Two classifiers A and B both achieve 90% accuracy on a ten-class test set. On every example that A classifies correctly, it assigns probability 0.91 to the correct class. On every example that B classifies correctly, it assigns 0.51.

   1. Compute the average cross-entropy loss of each classifier on its correctly classified examples. Why would knowing only that B's *average* correct-class probability is 0.51 not have sufficed to compute B's loss?
   2. Explain why these numbers do not decide which classifier is safer to deploy, and list the further information about the two classifiers you would need.

4. **Class-weighted loss.** In the screening example of Section 3.5.1, missing a sick patient costs much more than a false alarm. Give each class j a weight $w_j > 0$ and train f on the class-weighted average loss

$$\frac{1}{n}\sum_{i=1}^{n} w_{y^{(i)}}\, l\big(f(\mathbf{x}^{(i)}), y^{(i)}\big), \tag{3.25}$$

   where l is the cross-entropy loss and $y^{(i)}$ is the class of example i, so that every example of class j counts w_j times.

   1. Let the training examples be drawn with class prior $p(y)$ and class-conditional densities $p(\mathbf{x} \mid y)$. Show that the expected class-weighted loss is proportional to the expected unweighted loss under the distribution with the same class-conditionals and class prior proportional to $w_y\, p(y)$. Name the change to the training data, rather than to the loss, that has the same effect.
   2. Fix an input $\mathbf{x}$ with class probabilities $p(j \mid \mathbf{x})$, and suppose the model can output any distribution $\hat{\mathbf{y}}$. Show that the expected weighted loss

$$-\sum_{j} w_j\, p(j \mid \mathbf{x}) \log \hat{y}_j \tag{3.26}$$

      is minimized by $\hat{y}_j \propto w_j\, p(j \mid \mathbf{x})$.
   3. For the two classes healthy and sick, find the threshold on $p(\text{sick} \mid \mathbf{x})$ above which the arg max of this $\hat{\mathbf{y}}$ is sick, and choose the weights so that the threshold equals that of Equation 3.23. Explain why choosing $w_{\text{sick}} > w_{\text{healthy}}$ raises recall, and what it does to the number of false positives.

5. **Calibration and usefulness.** A ten-class problem has the same number of examples in every class.

    1. A classifier ignores its input and outputs the probability 0.1 for every class, breaking ties at random. Show that it is calibrated, and compute its accuracy.
    2. Another classifier is right on 95% of the examples and always reports confidence 1. Draw its reliability diagram and compute its expected calibration error.
    3. Explain why calibration and accuracy measure different things, and why neither can replace the other when a decision uses the threshold Equation 3.23.

6. code **ROC curves.** A binary classifier that outputs a score $s \in (0, 1)$ for the positive class can be thresholded at any $\tau \in [0, 1]$, not just $\frac{1}{2}$: predict positive whenever $s > \tau$. The *receiver operating characteristic* (ROC) curve plots the true-positive rate (recall) against the false-positive rate $\mathrm{FP}/(\mathrm{FP} + \mathrm{TN})$ as τ sweeps from 1 down to 0. Simulate 1000 positive and 1000 negative examples with scores $s = \sigma(z)$, where $\sigma(z) = 1/(1+\exp(-z))$ is the sigmoid, $z \sim \mathcal{N}(\mu, 1)$ for positives, and $z \sim \mathcal{N}(0, 1)$ for negatives.

    1. Compute and plot the ROC curve for $\mu \in \{0, 1, 2\}$, and state which points of the plot the endpoints $\tau = 1$ and $\tau = 0$ give.
    2. Show that a classifier whose scores are independent of the labels traces the diagonal in expectation.
    3. For $\mu = 2$, plot the *precision-recall curve*, precision against recall as τ sweeps, once with equal numbers of positives and negatives and once with positives making up 1% of the examples (keep 1000 negatives and draw 10 positives, or reweight). Explain why the ROC curve does not change with the prevalence while the precision-recall curve drops, and which of the two describes the screening test of this section better.

7. code **AUC as ranking probability.** ⊠ This problem uses the ROC curves and simulated scores of problem 6.

    1. Show that, when no two scores tie, the area under the ROC curve equals the probability that a randomly chosen positive example receives a higher score than a randomly chosen negative one, which makes it a threshold-free summary of ranking quality.
    2. Compute this probability in closed form for the simulated scores and compare it with the trapezoidal area under each of the curves.
    3. Apply the ROC curve and the AUC to the trained softmax regression on the validation images of two classes a and b, with a as the positive class and the score $s = \sigma(o_a - o_b)$, the sigmoid of the difference of the two logits: sneaker versus sandal, and shirt versus T-shirt. Compare the ordering of the two AUCs with the entries of the confusion matrix $\mathbf{C}$ of Section 3.3 for the same pairs, and explain why the two diagnostics need not agree.

8. code **Temperature scaling.** The text fits T on the first half of the validation images.

    1. Draw the reliability diagrams of the second half before and after scaling, and show that the accuracy does not change.
    2. Fit T on the training images instead and compare it with the T of the text. Explain why the fit should use data that the model was not trained on.
    3. Compute the expected calibration error of the unscaled model with 5, 10, and 20 bins. How much does the number depend on the binning?

3.6 Generalization in Classification

The softmax regression of Section 3.4 labels about 83% of its 10,000 validation images correctly. We would like to read that number as the accuracy the model will have on clothing images it has never seen, and two things stand in the way. First, the 10,000 images are a sample: another 10,000 from the same source would give a somewhat different accuracy. Second, a high score on data the model has seen proves little. A sufficiently expressive model can attain perfect training accuracy by memorizing the label of each training input and returning that label whenever the same input appears again. Memorization provides no rule for a new input, on which such a model can only guess.

This raises three questions:

1. How many test examples do we need to give a good estimate of the accuracy of our classifiers on the underlying population?
2. What happens if we keep evaluating models on the same test set repeatedly?
3. Why should fitting our linear models to the training set fare any better than memorizing it?

A word on names. The 10,000 images are Fashion-MNIST's test split, and this section speaks of them as a test set, as the theory does. Since we chose the learning rate and the number of epochs by looking at them, the 83% is strictly a validation score in the sense of Section 2.6; the answer to the second question says what that shortcut costs, and exercise 5 repairs it.

Section 2.6 estimated the error of a model on new data with held-out examples and kept the test set out of model selection. This section makes both ideas quantitative for classification. The first two questions have precise answers in terms of the size of the test set and the number of times we consult it. The third asks for a guarantee that holds before we see any test data, whatever the distribution of the data. Such guarantees explain why fitting a linear model differs from memorizing. For deep networks they are far too loose to be useful, which is why deep learning relies on test sets, and on the answers to the first two questions.

3.6.1 The Test Set

A test set measures one classifier. We first ask how precise that measurement is, as a function of the number of test examples.

3.6.1.1 Empirical and Population Error

Consider a fixed classifier f, without worrying about how it was obtained, and a *fresh* dataset $\mathcal{D} = \{(\mathbf{x}^{(i)}, y^{(i)})\}_{i=1}^{n}$ of examples that played no part in obtaining f. The *empirical error* of f on $\mathcal{D}$ is the fraction of examples whose label it gets wrong:

$$\epsilon_{\mathcal{D}}(f) = \frac{1}{n} \sum_{i=1}^{n} \mathbf{1}\left(f(\mathbf{x}^{(i)}) \neq y^{(i)}\right). \tag{3.27}$$

The *population error* is the probability that f gets the label wrong on a new example drawn from the distribution $P(X, Y)$ that generates the data:

$$\epsilon(f) = \mathbb{E}_{(\mathbf{x},y)\sim P}\left[\mathbf{1}\left(f(\mathbf{x}) \neq y\right)\right] = P\left(f(X) \neq Y\right). \tag{3.28}$$

These are the empirical risk and the risk of Section 2.6 for the *0–1 loss*, which charges 1 for a wrong label and 0 for a correct one. We want $\epsilon(f)$, but since we do not know P, we can compute only $\epsilon_{\mathcal{D}}(f)$.

Each term $\mathbf{1}(f(\mathbf{x}^{(i)}) \neq y^{(i)})$ equals 1 with probability $\epsilon(f)$ and 0 otherwise, like a coin that comes up heads with probability $\epsilon(f)$. The test examples are drawn independently, and f was fixed before they were drawn, so $\epsilon_{\mathcal{D}}(f)$ is the frequency of heads in n independent tosses of this coin. Two facts from Section 1.6 follow. The estimate is unbiased: $\mathbb{E}[\epsilon_{\mathcal{D}}(f)] = \epsilon(f)$. And the average of n independent draws has a standard error equal to the standard deviation of a single draw divided by $\sqrt{n}$. A variable that equals 1 with probability ϵ and 0 otherwise has variance $\epsilon(1-\epsilon)$, so the standard error of $\epsilon_{\mathcal{D}}(f)$ is $\sqrt{\epsilon(f)(1-\epsilon(f))/n}$.

For a classifier with population error 0.15, about that of softmax regression on Fashion-MNIST, and $n = 10{,}000$ test images, the standard error is $\sqrt{0.15 \cdot 0.85/10{,}000} \approx 0.0036$, about a third of a percentage point. The product $\epsilon(1-\epsilon)$ is largest at $\epsilon = 1/2$, where it equals $1/4$, so the standard error never exceeds $0.5/\sqrt{n}$, whatever the classifier. Halving it takes four times as many test examples. More test data does not change the error of the classifier, only our uncertainty about it.

3.6.1.2 Confidence Intervals

A standard error tells us how large deviations typically are, but not how often a deviation of a given size occurs. The *central limit theorem* fills this gap: when n is large, the average of n independent draws of a random variable with finite variance is approximately normally distributed, with the true mean as its mean and the standard error as its standard deviation (Section 36.3.2.3 states the theorem precisely). A normal random variable lies within 1.96 standard deviations of its mean with probability 0.95. For the test error this gives

$$P\left(|\epsilon_{\mathcal{D}}(f) - \epsilon(f)| \leq 1.96\sqrt{\frac{\epsilon(f)(1-\epsilon(f))}{n}}\right) \approx 0.95. \tag{3.29}$$

Turned around, the interval of 1.96 standard errors around $\epsilon_{\mathcal{D}}(f)$ contains the population error for 95% of test sets. It is called a 95% *confidence interval*, and we call its half-width the *radius*. The radius contains the unknown $\epsilon(f)$, of course; in practice $\epsilon_{\mathcal{D}}(f)$ takes its place. For softmax regression on the 10,000 Fashion-MNIST test images the radius is about $1.96 \cdot 0.0036 \approx 0.007$: its accuracy is known to within about 0.7 percentage points.

How many test examples give a radius of 0.01 at 95%, whatever the error of the classifier? In the worst case $\epsilon = 1/2$ we need $1.96 \cdot 0.5/\sqrt{n} \leq 0.01$, that is, $n \geq 9{,}604$. Many benchmark test sets have about 10,000 examples, Fashion-MNIST's among them, so the accuracy of a single model on them is known to within about one percentage point. An accuracy reported as 84.37% has more digits than such a test set supports.

3.6.1.3 Finite-Sample Guarantees

The central limit theorem describes the limit of large n. At any finite n the normal distribution only approximates the distribution of $\epsilon_{\mathcal{D}}(f)$, and the approximation is poor when the classifier makes few errors, for example when $\epsilon(f) = 0.001$ and $n = 1{,}000$, so that we expect a single error. Luckily, each term of the empirical error lies between 0 and 1, and

this alone suffices: an inequality due to Hoeffding (1963) gives a bound that holds at every n and for every distribution:

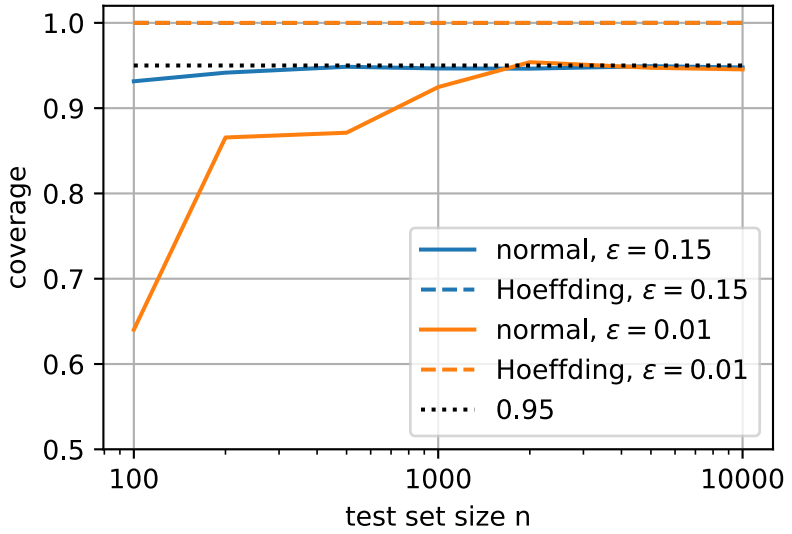
$$P\left(|\epsilon_{\mathcal{D}}(f) - \epsilon(f)| \geq t\right) \leq 2\exp\left(-2nt^2\right). \tag{3.30}$$

The probability that the test error misses the population error by t or more falls exponentially in nt^2 (Section 36.9 proves the inequality). Setting the right-hand side to $\delta = 0.05$ and solving for t gives the 95% *Hoeffding radius* $\sqrt{\log(2/\delta)/(2n)}$. A radius of 0.01 then requires $n \geq \log(40)/(2 \cdot 0.01^2) \approx 18{,}445$ examples, roughly 18,500, about twice the 9,604 of the normal approximation. Hoeffding's bound is more conservative because it holds at every n and uses only the range of each term, not its variance.

3.6.1.4 Simulated Test Sets

Does each interval contain the population error as often as it promises? We check by simulation, for two classifiers: one with population error $\epsilon(f) = 0.15$, close to that of our softmax regression, and one with $\epsilon(f) = 0.01$, which rarely errs. For each of several test set sizes n we draw 10,000 test sets. The number of errors on a test set is a sum of n independent 0–1 variables, which has a binomial distribution, so we draw it directly. For each test set we form the interval of the normal approximation, with $\epsilon_{\mathcal{D}}(f)$ in place of the unknown $\epsilon(f)$ as in practice, and the Hoeffding interval, and we record the fraction of test sets whose interval contains $\epsilon(f)$, its *coverage*.

```
rng = np.random.default_rng()
num_sets = 10000
ns = np.array([100, 200, 500, 1000, 2000, 5000, 10000])
curves, legend = [], []
for eps in [0.15, 0.01]:
    # The number of errors on a test set of size n is Binomial(n, eps)
    errs = [rng.binomial(n, eps, num_sets) / n for n in ns]
    normal = [1.96 * np.sqrt(e * (1 - e) / n) for e, n in zip(errs, ns)]
    hoeffding = [np.sqrt(np.log(2 / 0.05) / (2 * n)) for n in ns]
    for name, radii in [('normal', normal), ('Hoeffding', hoeffding)]:
        curves.append([np.mean(np.abs(e - eps) <= r)  # coverage
                       for e, r in zip(errs, radii)])
        legend.append(rf'{name}, $\epsilon = {eps}$')
d2l.plot(ns, curves + [[0.95] * len(ns)], 'test set size n', 'coverage',
         legend=legend + ['0.95'], xscale='log', ylim=(0.5, 1.02),
         fmts=('C0-', 'C0--', 'C1-', 'C1--', 'k:'), figsize=(4.5, 3))
```



At $\epsilon(f) = 0.15$ the normal interval contains the population error for 93% to 95% of the test sets at every size, close to its promise. At $\epsilon(f) = 0.01$ it falls short: on 100 test examples it covers only about 63% of the test sets, since a test set with no errors reports the interval $[0, 0]$, and it needs about 2,000 examples, some 20 expected errors, to reach 95%. The Hoeffding interval covers nearly all test sets at both error rates and every size. It keeps its promise without our checking whether an approximation holds, and the cost of that certainty is width: the Hoeffding interval is about twice as wide as the normal one, and at $\epsilon = 0.15$ the ratio of the two radii is 1.94 for every n.

3.6.2 Test Set Reuse

The guarantees so far concern one classifier, fixed before the test set was drawn. In practice a test set is consulted again and again: to compare architectures, to tune hyperparameters, and, for a public benchmark, by every paper that reports on it. Choosing the best of many classifiers inflates its score, and when the scores guide the design of the next classifier, the test set is no longer independent of the classifiers it evaluates.

3.6.2.1 Selection Among Many Classifiers

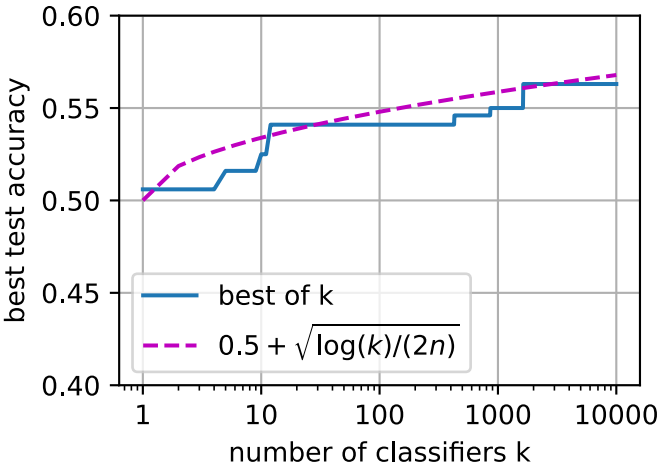
Suppose that we evaluate k classifiers $f_1, \dots, f_k$, all fixed before the test set is drawn, and report the one with the lowest test error. On its own, each test error lies within the Hoeffding radius with probability at least 0.95. The probability that at least one of them misses by more grows with k, and the classifier with the lowest test error is likely to be one whose error was underestimated. To guarantee all k estimates at once, we use the *union bound*: the probability that at least one of several events occurs is at most the sum of their probabilities. We apply it to the k events "the test error of f_j misses by t or more", each of which has probability at most $2\exp(-2nt^2)$ by Equation 3.30:

$$P\left(\max_{j=1,\dots,k} |\epsilon_{\mathcal{D}}(f_j) - \epsilon(f_j)| \geq t\right) \leq 2k \exp\left(-2nt^2\right). \tag{3.31}$$

Setting the right-hand side to δ shows that, with probability at least $1-\delta$, all k test errors lie within $\sqrt{\log(2k/\delta)/(2n)}$ of their population errors simultaneously. The radius grows only with the logarithm of k. On $10{,}000$ test examples at $\delta = 0.05$ it grows from 0.014 for one classifier to 0.023 for a thousand and 0.030 for a million.

To see the selection effect at work, we take the purest case: a test set of $n = 1{,}000$ examples whose binary labels are coin flips, and $k = 10{,}000$ classifiers that ignore their inputs and guess each label by a coin flip of their own. Every classifier has population accuracy 0.5, and no classifier can do better, because the labels carry no information about the inputs. We plot the best test accuracy among the first k classifiers as k grows, together with the reference curve $0.5 + \sqrt{\log(k)/(2n)}$.

```
n, k = 1000, 10000
labels = rng.random(n) < 0.5            # one test set with coin-flip labels
guesses = rng.random((k, n)) < 0.5      # k classifiers that guess at random
accuracy = (guesses == labels).mean(axis=1)
ks = np.arange(1, k + 1)
d2l.plot(ks, [np.maximum.accumulate(accuracy),
              0.5 + np.sqrt(np.log(ks) / (2 * n))],
         'number of classifiers k', 'best test accuracy',
         legend=['best of k', r'$0.5 + \sqrt{\log(k) / (2n)}$'], xscale='log',
         ylim=(0.4, 0.6))
```



The best test accuracy climbs to about 0.56 after 10,000 classifiers and stays close to the curve, mostly just below it. Why this curve? Each of the two directions of the deviation in Equation 3.30 has probability at most $\exp(-2nt^2)$, so the best of k accuracies exceeds $0.5+t$ with probability at most $k\exp(-2nt^2)$, which equals 1 on the curve and falls quickly above it. No classifier learned anything; the climb comes from selection alone. Whenever we keep the best of many models by their score on one test set, part of its apparent advantage is of this kind, and the union bound says how large that part can be.

3.6.2.2 Adaptive Overfitting

The union bound requires the k classifiers to be fixed before the test set is drawn. Model development rarely works that way: we train a model, look at its test score, and let the score guide the next model. Each new model then depends on the test set, and the bound no longer applies. The test score of a model chosen with the help of earlier scores on the same test set can overestimate its population accuracy far more than selection alone does. This is called *adaptive overfitting* (Dwork et al. 2015).

A simple strategy shows how large the effect can be, using nothing but test scores (Blum and Hardt 2015). We take the first 1,000 random guessers from above and keep those whose test accuracy exceeds 0.5. A new classifier then predicts, for each test input, the label that the majority of the kept guessers predict. On a new input, the kept guessers still flip coins, so their majority vote is again a coin flip, with population accuracy 0.5.

```
kept = guesses[:1000][accuracy[:1000] > 0.5]  # guessers that scored above 0.5
vote = kept.mean(axis=0) > 0.5                # their majority vote
print(f'majority vote: {(vote == labels).mean():.3f}, '
      f'best single guesser: {accuracy[:1000].max():.3f}')
```

```
majority vote: 0.712, best single guesser: 0.550
```

The majority vote scores about 0.7 on the test set, far above the best of the 1,000 guessers it combines (between 0.54 and 0.57) and the best of all 10,000 (about 0.56). It never saw a label. Each test score revealed a little about which guesses agree with the labels, and the vote added up this information over a thousand scores. Exercise 4 takes the idea to its end: about n well-chosen scores reveal every label of the test set.

The remedy is the division of labor of Section 2.6: a validation set absorbs model selection and the tuning of hyperparameters, and the test set is consulted once, for the final model. Public benchmarks are reused anyway. When Recht et al. (2019) collected new test sets for CIFAR-10 and ImageNet following the original procedures, accuracies dropped by 3 to 15 percentage points, but the models largely kept their ranking, and slight differences between the old and the new images explained the drops better than years of adaptive reuse. The worst case is therefore rare for large benchmarks, but real for small test sets that are queried often, such as the public leaderboards of competitions.

3.6.3 Statistical Learning Theory

A test set certifies a classifier after training. It does not explain why training should produce a classifier that generalizes, and it cannot tell us in advance how much training data a model needs. Statistical learning theory answers these questions by reasoning about the training error. This is a rather optimistic estimate: after all, training picked the classifier precisely because it made this number small. From the point of view of the training set, the learner is a modeler who evaluated every classifier it could choose on the same data and kept the best: selection among many classifiers, taken to the extreme.

We should say at the outset that this part is more abstract than the rest of the chapter and that its main result is stated without proof. It can be skipped on a first reading.

3.6.3.1 Uniform Convergence

Let $\mathcal{F}$ be the set of classifiers the learner can choose from, called its *hypothesis class*. We restrict ourselves to binary classifiers with labels ± 1, where the theory is simplest to state. For binary linear classification, $\mathcal{F}$ is the set of all classifiers $f(\mathbf{x}) = \operatorname{sign}(\mathbf{w}^\top \mathbf{x} + b)$. Let $\mathcal{S}$ be a training set of n examples, $\epsilon_{\mathcal{S}}(f)$ the empirical error of f on it, and $f_{\mathcal{S}}$ the classifier that the learner picks.

Let us start with a case that we can already handle, a finite class. The union bound Equation 3.31 applies with $k = |\mathcal{F}|$ and the training set in place of the test set, because the class, unlike $f_{\mathcal{S}}$, was fixed before the training data were drawn. With probability at least $1 - \delta$,

$$|\epsilon_{\mathcal{S}}(f) - \epsilon(f)| \le \sqrt{\frac{\log(2|\mathcal{F}|/\delta)}{2n}} \quad \text{for every } f \in \mathcal{F}. \tag{3.32}$$

Because the bound holds for every member of the class at once, it holds in particular for $f_{\mathcal{S}}$, however the learner made its choice. This is the key step. A guarantee of this kind is called *uniform convergence*. It makes the training error a trustworthy estimate of the population error once n is large compared with $\log|\mathcal{F}|$.

Unfortunately, the bound does not yet apply to linear classifiers. Their weights and bias are real numbers, so $\mathcal{F}$ is infinite, and so is the right-hand side. Could a better proof remove the size of the class from the bound altogether? Memorization shows that it could not. A memorizer has training error 0 on every training set, yet when the labels are coin flips its population error is $1/2$, so no bound on its gap can shrink as n grows. The way forward is to count not the classifiers but the different ways in which they can label the data.

3.6.3.2 The VC Dimension

Two classifiers that label the training points alike have the same training error; as far as the training set can tell, they are one classifier. And on n points a class of binary classifiers can produce at most 2^n labelings, however many classifiers it contains. A class *shatters* a set of points if it produces all 2^n labelings: for every assignment of labels ± 1 to the points, some $f \in \mathcal{F}$ agrees with it. The *Vapnik–Chervonenkis dimension*, or *VC dimension*, of $\mathcal{F}$ is the size of the largest set of points that $\mathcal{F}$ shatters (Vapnik and Chervonenkis 1971). It measures flexibility by the number of points on which the class can fit arbitrary labels. A memorizer shatters every finite set of distinct points, so its VC dimension is infinite.

It is instructive to work out the VC dimension of lines in the plane (Figure 3.6). Three points that do not lie on a line can be shattered: the figure shows a separating line for each of their eight labelings. Four points cannot, wherever we place them, because a halfplane that contains some points also contains the segments and triangles they span. If the four points form a convex quadrilateral, no line gives the ends of each diagonal their own label, since the diagonals cross. If one point lies inside the triangle of the other three, no line separates it from them, and if three lie on a line, no line separates the middle one from the outer two. The VC dimension of lines in the plane is therefore 3.

The same holds in every dimension: linear classifiers $\operatorname{sign}(\mathbf{w}^\top \mathbf{x} + b)$ on $\mathbb{R}^d$ have VC dimension $d + 1$. That some $d + 1$ points can be shattered is easy to see. Take $\mathbf{0}, \mathbf{e}_1, \dots, \mathbf{e}_d$, the origin and the standard unit vectors: the bias b alone decides the label of $\mathbf{0}$, and then each weight w_i decides the label of $\mathbf{e}_i$. That no $d + 2$ points can be shattered takes more work,

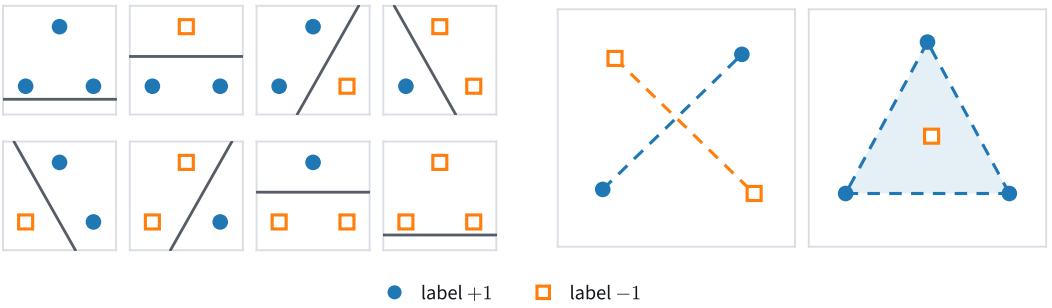


Figure 3.6: Lines in the plane shatter three points but no four. Left: a separating line for each of the $2^3 = 8$ labelings of three points not on a line. Middle and right: the two ways four points can lie when no three are on a line, each with a labeling that no line produces. The dashed diagonals of a convex quadrilateral cross, and a halfplane that contains the corners of a triangle contains every point inside it.

which we leave to Exercise 6. For linear classifiers the VC dimension thus equals the number of parameters. It is tempting to take this for a rule, and it is not one: the classifiers $\text{sign}(\sin(\omega x))$ on $\mathbb{R}$ have the single parameter ω and infinite VC dimension.

3.6.3.3 The VC Bound

What does a finite VC dimension give us? It limits how fast the number of labelings grows with the number of points. For points of which no three lie on a line, lines in the plane produce all 8 labelings of three points, 14 of the 16 labelings of four, and, in general, $n^2 - n + 2$ of the 2^n labelings of n points: 92 of 1,024 for ten points, and fewer than 10^4 of about 10^{30} for a hundred. Beyond the VC dimension the count grows polynomially in n, not exponentially. *Sauer's lemma* states this for every class: a class of VC dimension v produces at most $(en/v)^v$ labelings of any $n \geq v$ points.

Vapnik and Chervonenkis (1971) showed that the union bound can be taken over these labelings instead of over the classifiers. This is the hard step of the theory, and we report its outcome without proof (Section 36.10 gives one). The term $\log|\mathcal{F}|$ in Equation 3.32 is replaced by roughly $\log\left((en/v)^v\right) = v\log(en/v)$, and a finer analysis removes the logarithm. The result is the *VC bound*: there is a constant c, the same for every class and every distribution, such that with probability at least $1-\delta$ over the draw of the training set,

$$\epsilon(f) \leq \epsilon_{\mathcal{S}}(f) + c\sqrt{\frac{v + \log(1/\delta)}{n}} \quad \text{for every } f \in \mathcal{F}. \tag{3.33}$$

The population error of every classifier in the class, the trained one included, exceeds its training error by at most a term that shrinks like $\sqrt{v/n}$. This is the rate $1/\sqrt{n}$ of a single test set, and the class enters only through its VC dimension.

This answers the third question. A linear classifier on d features has VC dimension $d+1$, so once n is large compared with d, a small training error carries over to new data, whatever their distribution. A memorizer has infinite VC dimension, and the bound says nothing about it, as it must.

How much does the guarantee promise in numbers? Not much, it turns out. The bound treats binary classifiers, whereas our model tells ten classes apart, so let us take a binary problem on the same images, say telling shirts from everything else. With $d = 784$ pixels as features and $n = 60{,}000$ training images, the square-root term alone is $\sqrt{785/60{,}000} \approx 0.11$, before the constant c multiplies it, and for ten classes the bound only gets looser. Now set this beside what happened. In Section 3.3 the model reached about 84% on its training images and about 83% on the test images: a gap of one or two points, where the bound allows $0.11c$. The bound has to cover the worst distribution the data could have, and real data are rarely that adversarial. A test set of 10,000 images, by comparison, pins the error of the trained classifier down to about ± 0.01. This is how to hold a worst-case bound: it tells us which quantities govern the gap, v and n, and not how large the gap will be.

3.6.3.4 Limits of Worst-Case Bounds

For deep networks the VC bound guarantees nothing at all. Deep networks can fit essentially any labeling of their training set, random labels included (Zhang et al. 2021), so their VC dimension is at least the size of the training set. Yet, trained on real labels, the same networks generalize well, and often better as they grow larger. Bounds that depend on the data and on the training algorithm, not only on the class, are an active area of research. Section 36.10 proves the finite-class bound of this section and develops one such tool, Rademacher complexity, which measures how well a class can fit random signs on the actual data. None of these bounds is sharp yet. Why deep networks generalize at all is the question that Section 4.5 takes up.

3.6.4 Summary

The test error of a classifier fixed before the test set is drawn is an unbiased estimate of its population error, with a standard error of at most $0.5/\sqrt{n}$. About 10,000 test examples determine it to within ± 0.01 at 95% confidence by the central limit theorem, and about 18,500 do so with Hoeffding's inequality, which holds at every n. Reusing a test set weakens these guarantees: keeping the best of k fixed classifiers inflates its score by up to about $\sqrt{\log(k)/(2n)}$, and choices guided by earlier scores can inflate it much more, so model selection belongs on a validation set. Statistical learning theory extends the union bound from k classifiers to a whole class by counting labelings: the VC dimension, the size of the largest set of points a class shatters, takes the place of $\log|\mathcal{F}|$. Binary linear classifiers on $\mathbb{R}^d$ have VC dimension $d + 1$, so their training error approaches their population error at the rate $\sqrt{d/n}$. No memorizer has this guarantee, and for deep networks it is vacuous, so their generalization is certified with held-out data.

3.6.5 Exercises

1. **Two models, one test set.** A paper reports that model A reaches 91.2% and model B 91.6% on a test set of 10,000 images.
    1. Compute the standard error of each accuracy and the 95% radius of the normal approximation. Is the difference of 0.4 points meaningful if the two models were evaluated on two independent test sets of this size?
    2. Both models were in fact evaluated on the same 10,000 images. Explain why the question is now about the images on which the two models disagree, and describe

how you would decide it if the paper reported, for every test image, which models got it right.
3. How large a test set would make a 0.4-point difference between two independently evaluated models lie outside the radius?

2. **Two confidence radii.** Two radii r are meant to make the interval $\epsilon_{\mathcal{D}}(f) \pm r$ contain the population error ϵ of a classifier with probability 95%: the radius of the normal approximation $1.96\sqrt{\epsilon(1-\epsilon)/n}$ and the Hoeffding radius $\sqrt{\log(2/0.05)/(2n)}$.
   1. The standard error $\sqrt{\epsilon(1-\epsilon)/n}$ of $\epsilon_{\mathcal{D}}(f)$ is exact for every n. Which ingredient of the first interval is then only an asymptotic approximation?
   2. Show that the ratio of the Hoeffding radius to the first radius does not depend on n, and compute it as a function of ϵ. For which ϵ is Hoeffding's bound least conservative, and by how much does it then exceed the first radius?

3. **No errors on the test set.** A classifier makes no error on a test set of n examples.
   1. State the interval that the normal approximation, with $\epsilon_{\mathcal{D}}(f)$ in place of $\epsilon(f)$, reports, and the Hoeffding interval.
   2. Show that a classifier with population error ϵ makes no error on n independent test examples with probability $(1-\epsilon)^n \le e^{-n\epsilon}$. Conclude that if $\epsilon \ge 3/n$, a test set without errors occurs with probability below 0.05 (the "rule of three").
   3. Compare $3/n$ with the Hoeffding radius for $n = 100$ and $n = 10{,}000$, and explain the difference.

4. **Leaking a test set.** Somebody else holds a test set $\mathcal{D}$ of n examples with binary labels and releases only the inputs. You may submit any model f, with no restriction on the model class, and you receive only its test error $\epsilon_{\mathcal{D}}(f)$.
   1. Give a strategy that recovers every test label, and hence a model with test error 0 whatever its population error, and count the submissions it uses.
   2. Each answer $\epsilon_{\mathcal{D}}(f)$ takes one of $n+1$ values. Use this to lower-bound the number of submissions that any strategy needs to distinguish all 2^n labelings.

5. code **A validation split for Fashion-MNIST.** The chapter selected models on the 10,000 test images. Hold out 10,000 of the 60,000 training images as a validation split instead, and train softmax regression on the remaining 50,000.
   1. Choose the learning rate from $\{0.01, 0.03, 0.1, 0.3, 1\}$ by its accuracy on the validation split.
   2. Evaluate the selected model once on the test images, and compare its test accuracy with its validation accuracy. Is the difference within the radius of the 95% interval for 10,000 examples?
   3. Explain why the test accuracy, measured once after the choice, is an unbiased estimate for the selected model, while the best validation accuracy is not.

6. **VC dimension of linear classifiers.** ⊠ Consider the linear classifiers $f(\mathbf{x}) = \text{sign}(\mathbf{w}^\top \mathbf{x} + b)$ on $\mathbb{R}^d$.
   1. Show that they shatter the $d+1$ points $\mathbf{0}, \mathbf{e}_1, \dots, \mathbf{e}_d$. Given arbitrary labels $y_0, y_1, \dots, y_d \in \{\pm 1\}$, construct b and $\mathbf{w}$ that produce them. Hint: $f(\mathbf{0})$ depends on b alone, so choose b first.
   2. Let $\mathbf{x}_1, \dots, \mathbf{x}_{d+2} \in \mathbb{R}^d$. Show that the $d+2$ vectors $[\mathbf{x}_i, 1] \in \mathbb{R}^{d+1}$, each $\mathbf{x}_i$ with a 1 appended, are linearly dependent, so that $\sum_i a_i [\mathbf{x}_i, 1] = \mathbf{0}$ for some a_i that

are not all zero. Let I be the indices with $a_i > 0$ and J those with $a_i < 0$. Show that neither is empty and that no linear classifier labels the points in I with $+1$ and those in J with -1. Hint: evaluate $\sum_i a_i(\mathbf{w}^\top \mathbf{x}_i + b)$ in two ways.

3. Conclude that the VC dimension is $d + 1$, and show that the convex hulls of $\{\mathbf{x}_i : i \in I\}$ and $\{\mathbf{x}_j : j \in J\}$ intersect.

7. **Collinear points.** In Figure 3.6 the three shattered points do not lie on a line. Show that three points on a line cannot be shattered by halfplanes: which labeling is impossible? Explain why this does not contradict the VC dimension of lines in the plane being 3.

8. **VC dimension of polynomials.** What is the VC dimension of the class of classifiers $f(x) = \text{sign}(p(x))$ with p a polynomial of degree at most five in $x \in \mathbb{R}$? What is it for $\mathbf{x} \in \mathbb{R}^d$? Hint: such a classifier is a linear classifier on the monomials of $\mathbf{x}$. Then say what the VC bound predicts for the training error of a degree-five polynomial classifier on Fashion-MNIST, and why nobody trains one.

3.7 Environment and Distribution Shift

The test error of Chapter 3.6 tells us how well a classifier will do on new data, provided the new data come from the distribution of the test set. After deployment this condition often fails, even when the model has no influence on the data: a hospital buys a new scanner, an epidemic changes which diseases patients have, and the meaning of words drifts. The test set then gives no warning. Can we still predict, or even improve, how the model does on the data it now sees?

The model can also change the data itself. Suppose that a bank trains a model to predict which loan applicants will repay, and the model finds that applicants who wear Oxfords tend to repay while applicants who wear sneakers tend to default. The model scores well on held-out applications, so the bank approves everyone in Oxfords and declines everyone in sneakers. Applicants soon notice, everyone starts wearing Oxfords, and the fraction who repay stays what it was. The model was accurate on the data it was tested on and is useless on the data it now sees, because using it changed the data. This is an instance of *Goodhart's law*: when a measure becomes a target, it ceases to be a good measure.

We first show that nothing carries over from the old distribution to the new one unless we assume something about how the two are related, and we sort the common assumptions by which part of the joint distribution of inputs and labels they keep fixed. Under two of them, unlabeled examples from the new distribution suffice to correct training by reweighting the training examples; the third needs new labels. We end with models whose decisions change their own data, where statistical correction stops and questions of policy begin.

3.7.1 Types of Distribution Shift

Let us start by naming the two distributions. We train on labeled examples drawn from a *source distribution* with density $p_S(\mathbf{x}, y)$ and deploy the model on examples drawn from a *target distribution* with density $p_T(\mathbf{x}, y)$, whose inputs we observe but, at least at first, not their labels.

Without an assumption that links p_S and p_T, the source data tell us nothing about the target. This is more than a formal caveat. Consider telling cats from dogs, and suppose that the

target keeps the inputs of the source, $p_T(\mathbf{x}) = p_S(\mathbf{x})$, but swaps the labels: whatever the source calls a cat, the target calls a dog, so that $p_T(y = \text{cat} \mid \mathbf{x}) = p_S(y = \text{dog} \mid \mathbf{x})$. The unlabeled target inputs have exactly the distribution of the source inputs, so no statistic computed from them can tell this world from one without any shift. Yet a classifier that is always right on the source is always wrong on the target. Every method in this section therefore starts from an assumption about which part of the distribution stays fixed.

The joint density factors in two ways,

$$p(\mathbf{x}, y) = p(\mathbf{x})\, p(y \mid \mathbf{x}) = p(y)\, p(\mathbf{x} \mid y). \tag{3.34}$$

The first factorization describes data generated by drawing an input and then labeling it; the second describes data generated by drawing a class and then producing an input of that class. The three common kinds of shift each let one factor change and keep another fixed:

Shift	Changes	Stays fixed	Natural when
covariate shift	$p(\mathbf{x})$	$p(y \mid \mathbf{x})$	the input causes the label
label shift	$p(y)$	$p(\mathbf{x} \mid y)$	the label causes the input
concept shift	$p(y \mid \mathbf{x})$	often $p(\mathbf{x})$	the meaning of the label changes

Figure 3.7 draws the three cases for a scalar input and two classes.

3.7.1.1 Covariate Shift

Under *covariate shift* the distribution of the inputs changes while the labeling rule $p(y \mid \mathbf{x})$ stays the same. (Statisticians call the inputs covariates.) The assumption is natural when the input causes the label: an image shows a cat because a cat was photographed or drawn. A classifier trained on photographs of cats and dogs (Figure 3.8) and deployed on cartoons (Figure 3.9) faces covariate shift. A drawn cat is still a cat, but cartoons occupy a region of image space that the training data barely visit. Training fits the model where the source data lie, and errors where only the target lies barely affect the training loss.

3.7.1.2 Label Shift

Under *label shift* the class proportions $p(y)$ change while each class keeps its appearance, $p(\mathbf{x} \mid y)$. The assumption is natural when the label causes the input. Diseases cause symptoms: during an influenza epidemic the fraction of patients who have influenza rises, but the symptoms of influenza stay what they were. Similarly, a classifier trained on equally many images of every class may be deployed where some classes are far more common than others.

Label shift changes the labeling rule as well. By Bayes' theorem, $p(y \mid \mathbf{x}) \propto p(\mathbf{x} \mid y)\, p(y)$, so a classifier trained on the source has the source class proportions built into it; in the middle column of Figure 3.7 the boundary moves toward the class that became rarer. Correcting label shift requires estimating only one number per class, not a density over inputs with thousands or millions of dimensions. It is the easier of the two shifts to correct.

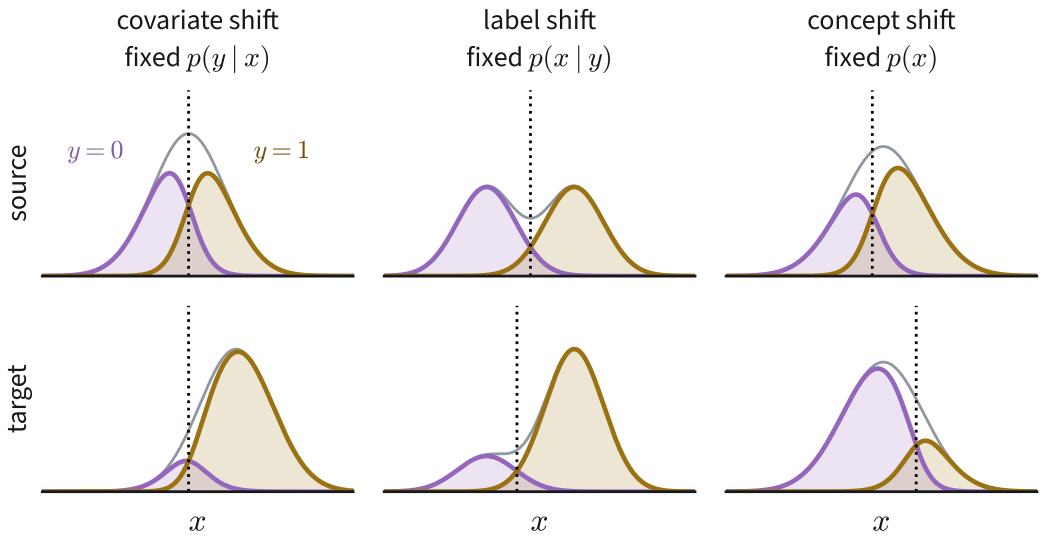


Figure 3.7: Three kinds of distribution shift for a scalar input x and two classes. Each panel plots $p(x, y = 0)$ and $p(x, y = 1)$, whose sum $p(x)$ is the gray outline; the top row is the source, the bottom row the target, and the dotted line marks the decision boundary, where both classes are equally likely. Covariate shift moves the inputs but not the boundary. Label shift keeps the shape of each class but changes the class proportions, which moves the boundary. Concept shift keeps $p(x)$ and moves the boundary.

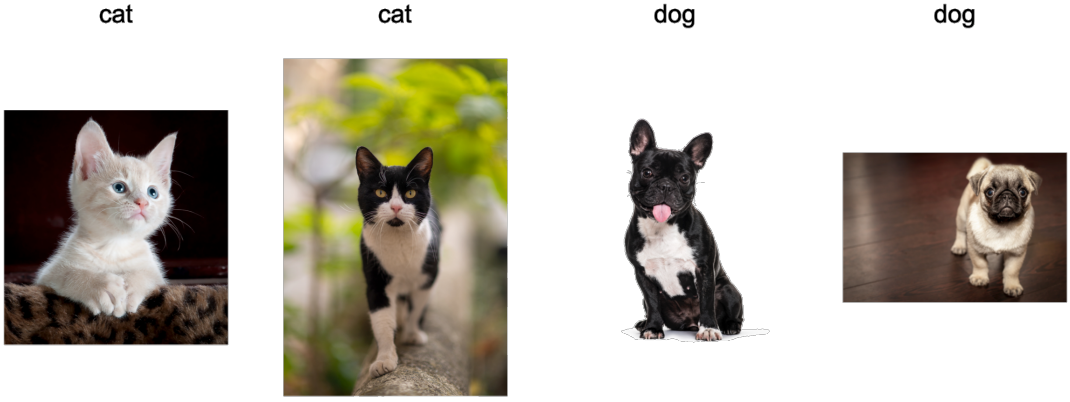


Figure 3.8: Source data for telling cats from dogs: photographs (illustrations: Lafeez Hossain / 500px / Getty Images; ilkermetinkursova / iStock / Getty Images Plus; GlobalP / iStock / Getty Images Plus; Musthafa Aboobakuru / 500px / Getty Images).

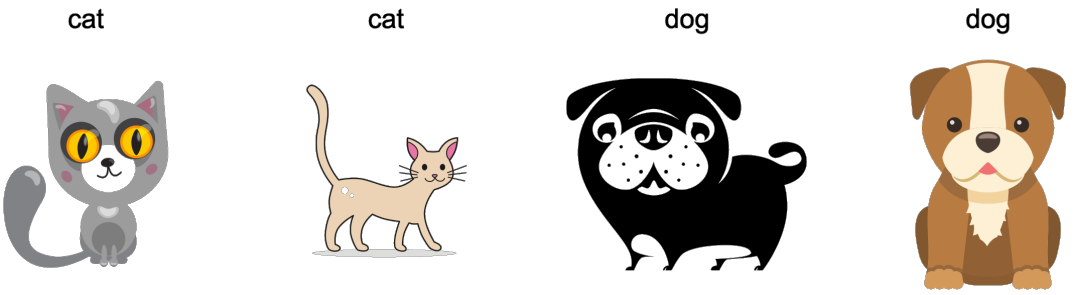


Figure 3.9: Target data for the same task: cartoons (illustrations: SIBAS_minich / iStock / Getty Images Plus; Ghrzuzudu / iStock / Getty Images Plus; id-work / DigitalVision Vectors / Getty Images; Yime / iStock / Getty Images Plus).

3.7.1.3 Concept Shift

Under *concept shift* the labeling rule itself changes: the same input receives a different label. Categories defined by people drift in this way. Diagnostic criteria for mental illnesses are revised every few years, so the same patient record can receive a different diagnosis. Labels also vary by place: the name for a sweetened carbonated drink depends on where in the United States one asks (Figure 3.10), so a system that names the drink in a photograph faces concept shift as it moves across the country.

Concept shift is often gradual. A recommender trained in winter that keeps suggesting Santa hats in January has missed a change in what users want given the same browsing history. When the inputs keep their distribution, as in the right column of Figure 3.7, unlabeled target data cannot reveal such a change: it is a partial version of the label swap with which we started.

3.7.2 Covariate Shift Correction

In Section 2.6.1 we defined the risk of a model f as its expected loss $l(f(\mathbf{x}), y)$ on a new example, estimated by the average loss on the training examples, the empirical risk. Under shift we must say which distribution the expectation is over. We care about the *target risk*

$$R_T[f] = \mathbb{E}_{(\mathbf{x},y)\sim p_T}\left[l(f(\mathbf{x}), y)\right], \tag{3.35}$$

the expected loss in deployment. The training examples are drawn from p_S, so their empirical risk estimates the source risk $R_S[f]$ instead. Under covariate shift, labeled source examples and unlabeled target inputs suffice to estimate R_T, and therefore to train for the target.

3.7.2.1 Importance Weights

Under covariate shift the source and the target share the labeling rule, which we write $p(y \mid \mathbf{x})$ without a subscript, and differ only in the density of the inputs. The trick is to multiply and divide by $p_S(\mathbf{x})$ inside the target risk, which turns it into an expectation over the source (Shimodaira 2000):

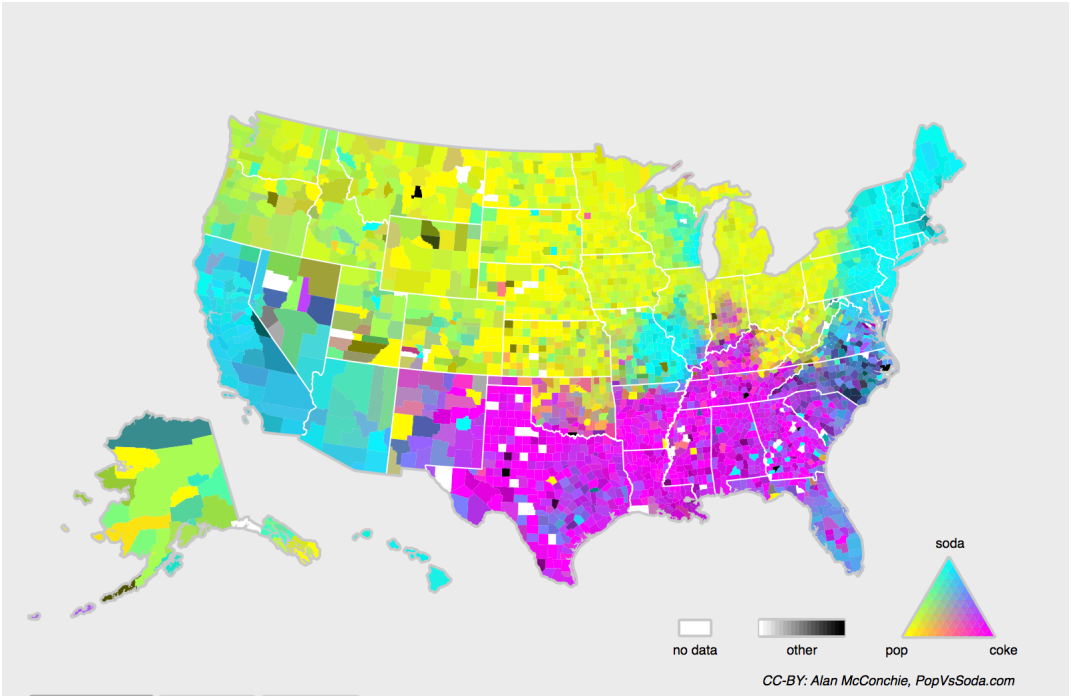


Figure 3.10: The most common name for soft drinks in each county of the United States: “pop” (yellow), “soda” (cyan), or “coke” (magenta) (CC-BY: Alan McConchie, PopVsSoda.com).

$$\begin{aligned} R_T[f] &= \sum_y \int l(f(\mathbf{x}), y)\, p(y \mid \mathbf{x})\, p_T(\mathbf{x})\, d\mathbf{x} \\ &= \sum_y \int l(f(\mathbf{x}), y)\, p(y \mid \mathbf{x})\, p_S(\mathbf{x})\, \frac{p_T(\mathbf{x})}{p_S(\mathbf{x})}\, d\mathbf{x} = \mathbb{E}_{(\mathbf{x},y)\sim p_S}\left[\beta(\mathbf{x})\, l(f(\mathbf{x}), y)\right], \end{aligned} \tag{3.36}$$

where

$$\beta(\mathbf{x}) = \frac{p_T(\mathbf{x})}{p_S(\mathbf{x})}$$

is the *importance weight* of the input $\mathbf{x}$. The last step uses the covariate shift assumption: $p(y \mid \mathbf{x})\, p_S(\mathbf{x})$ is the source density $p_S(\mathbf{x}, y)$ because the labeling rule is shared. The target risk is thus a source expectation in which each example counts in proportion to how much more likely its input is under the target than under the source. Replacing the expectation by an average over the n training examples gives *weighted empirical risk minimization*:

$$\underset{f}{\text{minimize}}\ \frac{1}{n}\sum_{i=1}^{n} \beta_i\, l(f(\mathbf{x}^{(i)}), y^{(i)}), \quad \text{where } \beta_i = \beta(\mathbf{x}^{(i)}). \tag{3.37}$$

For a number, take the running example of this section. The inputs lie in the plane; the source is a standard Gaussian centered at the origin, and the target is the same Gaussian centered at $(2, 0)$. Then

$$\beta(\mathbf{x}) = \frac{\exp\left(-\frac{1}{2}\|\mathbf{x} - (2,0)\|^2\right)}{\exp\left(-\frac{1}{2}\|\mathbf{x}\|^2\right)} = \exp(2x_1 - 2),$$

which depends only on the coordinate along which the distribution moved. An input at the origin, typical of the source, has weight $e^{-2} \approx 0.14$. An input at $(2, 0)$, typical of the target, has weight $e^2 \approx 7.4$, and an input at $(3, 0)$, which the source rarely produces, has weight $e^4 \approx 55$ (Figure 3.11). One such input counts as much as 400 at the origin.

The identity divides by $p_S(\mathbf{x})$, so it holds only if $p_S(\mathbf{x}) > 0$ wherever $p_T(\mathbf{x}) > 0$: every input that the target can produce must be possible, if rare, under the source. Where the target has mass and the source has none, there are no examples to reweight. A startup that some of us worked with ran into this problem. It developed a blood test for a disease of older men, using blood from patients with the disease and, as healthy controls, from university students. A classifier separated the two groups almost perfectly, but it could do so by age, hormone levels, and diet rather than by the disease. The test was meant for older men, and the training data contained no healthy older men; no weighting of the students' samples can stand in for them.

3.7.2.2 Estimating Weights with a Domain Classifier

The weights seem to require both densities, and estimating a density over images is much harder than classifying images. Luckily we need only their ratio, and a ratio is something a classifier can estimate. We pool the n source inputs with m unlabeled target inputs and

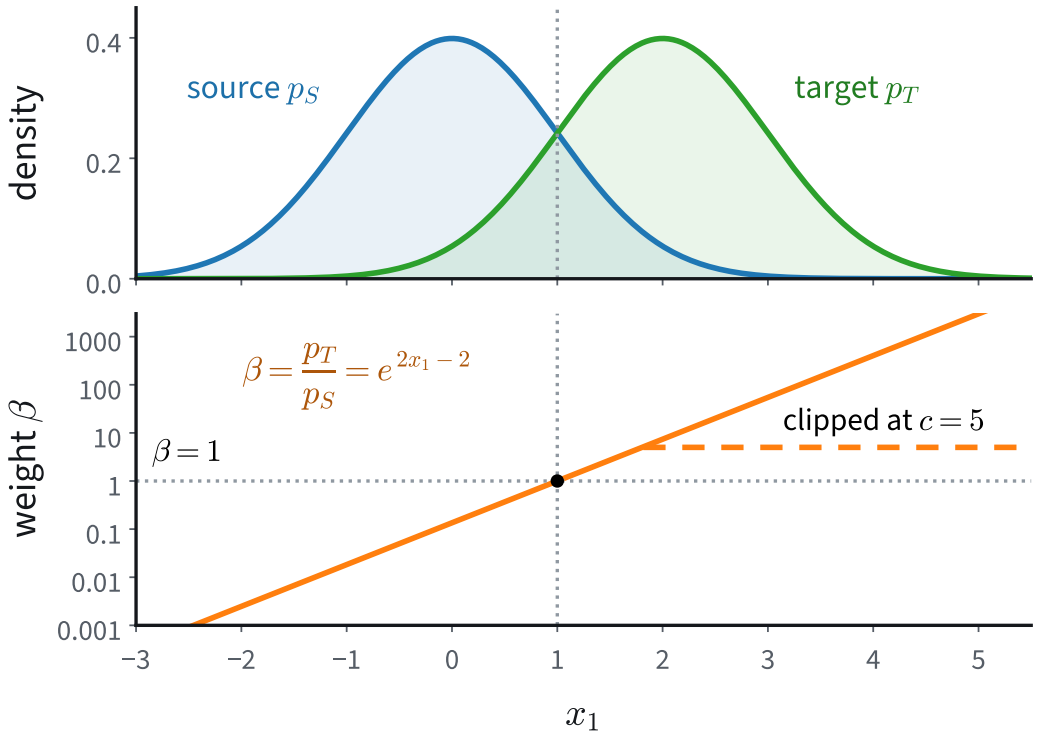


Figure 3.11: Importance weights for the running example along x_1. Top: the source density p_S, a standard Gaussian, and the target density p_T, centered at 2. Bottom, on a logarithmic scale: $\beta = p_T/p_S = e^{2x_1 - 2}$, which equals 1 where the densities cross and grows by a factor of $e^2 \approx 7.4$ per unit of x_1 where the source thins out. The dashed line clips the weight at $c = 5$.

give each a *domain label* z, with $z = 1$ for a target input and $z = 0$ for a source input. A classifier trained to predict z from $\mathbf{x}$ estimates $P(z = 1 \mid \mathbf{x})$. By Bayes' theorem, with prior probability $m/(n + m)$ for the target,

$$P(z = 1 \mid \mathbf{x}) = \frac{m\, p_T(\mathbf{x})}{m\, p_T(\mathbf{x}) + n\, p_S(\mathbf{x})}, \quad \text{so} \quad \frac{P(z = 1 \mid \mathbf{x})}{P(z = 0 \mid \mathbf{x})} = \frac{m}{n}\, \beta(\mathbf{x}). \tag{3.38}$$

The odds that an input came from the target are its importance weight, up to the ratio of the sample sizes: inputs that the classifier confidently assigns to the target get large weights.

Logistic regression models these odds directly. As we saw in Section 3.1, softmax regression with two classes predicts $P(z = 1 \mid \mathbf{x}) = \sigma(h(\mathbf{x}))$, where $\sigma(o) = 1/(1 + \exp(-o))$ is the logistic sigmoid and $h(\mathbf{x}) = \mathbf{w}^\top \mathbf{x} + b$ is the logarithm of the odds. The odds are therefore $\exp(h(\mathbf{x}))$, and

$$\beta(\mathbf{x}) = \frac{n}{m} \exp(h(\mathbf{x})). \tag{3.39}$$

The factor n/m multiplies all weights by the same number and does not change the minimizer of Equation 3.37, so we drop it. The domain classifier never sees a label y. The correction thus takes four steps:

1. Pool the source inputs, with $z = 0$, and the unlabeled target inputs, with $z = 1$.
2. Train a logistic regression classifier h to predict z from $\mathbf{x}$.
3. Give each training example the weight $\beta_i = \exp(h(\mathbf{x}^{(i)}))$, or the clipped weight $\beta_i = \min(\exp(h(\mathbf{x}^{(i)})), c)$ for a constant c.
4. Train the model f on the labeled source examples by minimizing the weighted empirical risk Equation 3.37.

Clipping limits the influence of the few examples with the largest weights, which make the weighted average noisy. How noisy? If the losses of the n examples are independent with equal variance, the weighted average $\sum_i \beta_i l_i / \sum_i \beta_i$ has the variance of an unweighted average over

$$n_{\text{eff}} = \frac{\left(\sum_{i=1}^{n} \beta_i\right)^2}{\sum_{i=1}^{n} \beta_i^2} \tag{3.40}$$

examples, its *effective sample size*. It equals n when all weights are equal and approaches 1 when a single weight dominates. Clipping the largest weights raises n_{eff} but introduces a bias: examples in the clipped region count less than the target distribution says they should.

3.7.2.3 A Two-Dimensional Example

We now run the four steps on the running example, where we know the true weights. Both domains share one labeling rule: $y = 1$ above the parabola $x_2 = \frac{1}{2}x_1^2 - 1$ and $y = 0$ below it, so this is covariate shift by construction. Since the boundary is curved, no linear classifier fits it everywhere: the best line near the origin, where the source lies, differs from the best line near $(2, 0)$, where the target lies. The computation needs only NumPy.

We draw 1,000 source examples with their labels and 1,000 target examples, whose labels we use only to measure accuracy. The generator is not seeded, so your numbers will differ from ours. How much they differ turns out to be part of the result.

```
rng = np.random.default_rng()
n = 1000
X_src = rng.normal(0, 1, (n, 2))            # source p_S: centered at (0, 0)
X_tgt = rng.normal(0, 1, (n, 2)) + [2, 0]   # target p_T: centered at (2, 0)
label = lambda X: (X[:, 1] > 0.5 * X[:, 0]**2 - 1).astype(float)
y_src, y_tgt = label(X_src), label(X_tgt)   # one labeling rule for both
```

The one learning algorithm we need is logistic regression, trained by gradient descent with an optional weight per example. By Section 3.1.2.2, the gradient of the cross-entropy loss with respect to the logit is the predicted probability minus the label, $\sigma(\mathbf{w}^\top \mathbf{x} + b) - y$, so the gradient with respect to $\mathbf{w}$ is this error times the input. A weight β_i scales the error of example i.

```
def fit_logreg(X, y, weights=None, lr=0.1, steps=2000):
    """Logistic regression by gradient descent on the weighted mean loss."""
    w, b = np.zeros(X.shape[1]), 0.0
    v = np.ones(len(y)) if weights is None else weights / weights.mean()
    for _ in range(steps):
        r = v * (1 / (1 + np.exp(-(X @ w + b))) - y)   # weighted error
        w -= lr * X.T @ r / len(y)
        b -= lr * r.mean()
    return w, b
```

Steps 1 and 2 train the domain classifier on the pooled inputs. For two Gaussians with the same covariance, the logarithm of the density ratio is linear in $\mathbf{x}$, here $2x_1 - 2$, so logistic regression can represent the true weights and we can check what it learns. The cell also reports the largest weight and the effective sample size, with and without clipping at $c = 5$.

```
z = np.concatenate([np.zeros(n), np.ones(n)])      # domain label: 1 = target
w_h, b_h = fit_logreg(np.concatenate([X_src, X_tgt]), z)
beta = np.exp(X_src @ w_h + b_h)                   # weights of source examples
ess = lambda v: v.sum()**2 / (v**2).sum()          # effective sample size
print(f'learned h(x) = {w_h[0]:.2f} x1 {w_h[1]:+.2f} x2 {b_h:+.2f}'
      f'   (true log-ratio: 2 x1 - 2)')
print(f'largest weight: {beta.max():.0f}; effective sample size: '
      f'{ess(beta):.0f} of {n}, {ess(np.minimum(beta, 5)):.0f} if clipped at 5')
```

```
learned h(x) = 2.00 x1 -0.05 x2 -2.01   (true log-ratio: 2 x1 - 2)
largest weight: 27; effective sample size: 116 of 1000, 219 if clipped at 5
```

The learned h is close to $2x_1 - 2$, so the domain classifier has found the true weights. They spread over several orders of magnitude, and the 1,000 source examples are worth far fewer: a few dozen, in an unlucky draw barely ten. Clipping at $c = 5$ raises the effective sample size to about 200. Rerun the cell and the largest weight jumps between a few dozen and several hundred, taking the unclipped effective sample size with it: a handful of points in the right tail of the source carry most of the weight, and how far that tail reaches is a matter of luck. Importance weights are high-variance quantities, and this is where we see it.

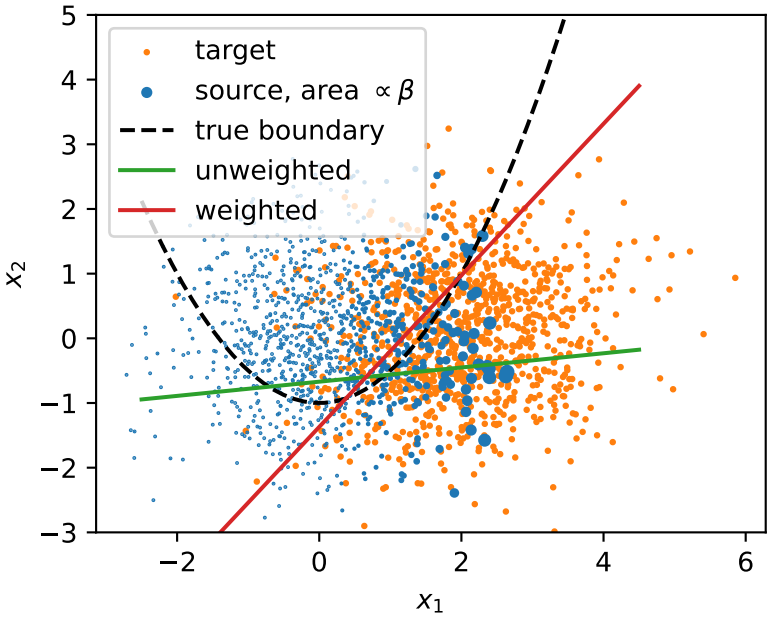
Step 4 trains the classifier three times on the labeled source data: without weights, with the weights β_i, and with the weights clipped at $c = 5$. We evaluate each model on the target and on the source, and compare with always predicting the most common target class.

```
acc = lambda wb, X, y: ((X @ wb[0] + wb[1] > 0) == (y == 1)).mean()
models = {name: fit_logreg(X_src, y_src, v) for name, v in [
    ('unweighted', None), ('weighted', beta), ('clipped', np.minimum(beta, 5))]}
for name, wb in models.items():
    print(f'{name:10s}  target accuracy {acc(wb, X_tgt, y_tgt):.3f}'
          f'   source accuracy {acc(wb, X_src, y_src):.3f}')
print(f'always predicting the most common target class: '
      f'{max(y_tgt.mean(), 1 - y_tgt.mean()):.3f}')
```

```
unweighted  target accuracy 0.581   source accuracy 0.859
weighted    target accuracy 0.950   source accuracy 0.836
clipped     target accuracy 0.949   source accuracy 0.839
always predicting the most common target class: 0.719
```

On the target, the unweighted model does worse than no model at all: it is right on only 45% to 65% of the examples, while always predicting the most common target class gets about 70%. Reweighting raises target accuracy to about 95% and lowers source accuracy by a few points. The model gives up a little where the source lies to gain a great deal where the target lies, as Equation 3.36 prescribes. Clipping changes almost nothing here, a point or two of target accuracy in either direction: a line has only three parameters, and even a dozen effective examples suffice to place it. Exercise 3 moves the target farther away. What did reweighting change? We plot the two decision boundaries with the data, drawing each source point with an area proportional to its weight.

```
x1 = np.linspace(-2.5, 4.5, 50)
d2l.set_figsize((4.5, 3.5))
d2l.plt.scatter(*X_tgt.T, s=2, c='C1', label='target')
d2l.plt.scatter(*X_src.T, s=np.clip(beta, 0.2, 20), c='C0',
                label=r'source, area $\propto \beta$')
d2l.plt.plot(x1, 0.5 * x1**2 - 1, 'k--', label='true boundary')
for name, fmt in [('unweighted', 'C2'), ('weighted', 'C3')]:
    (w1, w2), b = models[name]
    d2l.plt.plot(x1, -(w1 * x1 + b) / w2, fmt, label=name)
d2l.plt.xlabel('$x_1$'), d2l.plt.ylabel('$x_2$'), d2l.plt.ylim(-3, 5)
d2l.plt.legend();
```



The unweighted boundary is a nearly horizontal line at about $x_2 = -0.6$, the best line for the bottom of the parabola, where most source examples lie. Near $(2, 0)$ the parabola rises steeply, and this line assigns $y = 1$ to the many target points above it but below the parabola. The weighted boundary climbs to the right instead, with a slope of roughly 1 and sometimes steeper, along the right arm of the parabola. The large dots at the right edge of the source cloud determine it. The points near the origin, which is to say most of the sample, have almost no say.

3.7.3 Label Shift Correction

Under label shift it is $p(\mathbf{x} \mid y)$ that the two domains share. The argument that gave Equation 3.36, with the weight now on the label, gives

$$R_T[f] = \sum_y p_T(y) \int l(f(\mathbf{x}), y)\, p(\mathbf{x} \mid y)\, d\mathbf{x} = \mathbb{E}_{(\mathbf{x},y)\sim p_S}\left[\frac{p_T(y)}{p_S(y)}\, l(f(\mathbf{x}), y)\right]. \tag{3.41}$$

Each training example is weighted by $\beta_i = p_T(y^{(i)})/p_S(y^{(i)})$, the ratio of how common its class is in the target and in the source, and the model is again trained by weighted empirical risk minimization. The source proportions $p_S(y)$ are the class frequencies of the training labels. The target proportions $p_T(y)$ are the difficulty: the target comes without labels, so we cannot simply count.

3.7.3.1 Estimating the Target Class Proportions

Any classifier trained on the source can estimate the target proportions, provided we know how it errs. That knowledge is the column-normalized confusion matrix $\mathbf{C}$ of Section 3.3, with entries $C_{ij} = P(\hat{c} = i \mid y = j)$ for the predicted class $\hat{c}$, which we estimate on a labeled validation set from the source.

The key observation is that under label shift $\mathbf{C}$ is the same on the target: the prediction $\hat{c}$ depends only on $\mathbf{x}$, and the inputs of each class have the same distribution in both domains. Let μ_i be the fraction of target inputs that the classifier assigns to class i, which needs no labels. By the law of total probability,

$$\mu_i = \sum_{j=1}^{q} P(\hat{c} = i \mid y = j)\, p_T(y = j) = \sum_{j=1}^{q} C_{ij}\, p_T(y = j), \quad \text{that is,} \quad \mathbf{C}\,\mathbf{p}_T = \mu, \tag{3.42}$$

where the vectors $\mathbf{p}_T$ and μ collect the q target class proportions and prediction frequencies. This is the reason the confusion matrix of Section 3.3 keeps the true classes in its columns. The predictions we observe on the target are the true target proportions mixed by the classifier's errors, and if $\mathbf{C}$ is invertible, $\mathbf{p}_T = \mathbf{C}^{-1}\mu$ undoes the mixing.

Take the screening test of Section 3.5, which detects 90% of the sick and wrongly flags 5% of the healthy, and suppose it was validated on a source sample with equally many healthy ($y = 0$) and sick ($y = 1$) patients. With rows and columns ordered (healthy, sick),

$$\mathbf{C} = \begin{pmatrix} 0.95 & 0.10 \\ 0.05 & 0.90 \end{pmatrix}.$$

In the target population the test flags 13.5% of patients, $\mu = (0.865, 0.135)$. Writing p for the target fraction of sick patients, the second row of Equation 3.42 reads $0.05\,(1 - p) + 0.90\,p = 0.135$, so $p = 0.085/0.85 = 0.10$. The 13.5% flagged overstates the prevalence: of all patients, $0.05 \times 0.9 = 4.5\%$ are false alarms and only $0.10 \times 0.1 = 1\%$ are missed cases, and the solve trades the one against the other. Retraining then weights each sick training example by $0.1/0.5 = 0.2$ and each healthy one by $0.9/0.5 = 1.8$.

We check the estimator on Fashion-MNIST, with the softmax regression of Section 3.4 as the classifier. We train it as there and keep its logits and the labels of the 10,000 validation images as NumPy arrays.

```
data = d2l.FashionMNIST(batch_size=256)
model = d2l.SoftmaxRegression(num_outputs=10, lr=0.1)
trainer = d2l.Trainer(max_epochs=10)
trainer.board.display = False
trainer.fit(model, data)
O, y = [], []
for X, Y in data.val_dataloader():
    O.append(d2l.numpy(model(X)))
    y.append(d2l.numpy(Y))
O, y = np.concatenate(O), np.concatenate(y)
pred = O.argmax(axis=1)
```

Half of the validation images, chosen at random, serve as the labeled source validation set on which we estimate $\mathbf{C}$. The source is balanced: every class has the proportion 0.1. From the other half we draw a target sample in which half of the images are shirts: 400 shirts and 45 images of each other class. Its labels serve only to check the estimate.

```
perm = rng.permutation(len(y))
src, pool = perm[:5000], perm[5000:]  # Source validation set; target pool
C = np.zeros((10, 10))
np.add.at(C, (pred[src], y[src]), 1)  # The confusion matrix of the source
C /= C.sum(axis=0, keepdims=True)     # Each column sums to 1
counts = np.full(10, 45)
counts[6] = 400                       # Half of the target images are shirts
p_T = counts / counts.sum()
tgt = np.concatenate([rng.choice(pool[y[pool] == j], counts[j], replace=False)
                      for j in range(10)])
mu = np.bincount(pred[tgt], minlength=10) / len(tgt)  # Needs no labels
p_hat = np.linalg.solve(C, mu)
for j, name in enumerate(data.text_labels(range(10))):
    print(f'{name:10s} true {p_T[j]:.2f}, predicted {mu[j]:.3f}, '
          f'estimated {p_hat[j]:.3f}')
```

```
T-shirt    true 0.06, predicted 0.132, estimated 0.064
trouser    true 0.06, predicted 0.055, estimated 0.052
pullover   true 0.06, predicted 0.070, estimated 0.043
dress      true 0.06, predicted 0.060, estimated 0.047
coat       true 0.06, predicted 0.145, estimated 0.072
sandal     true 0.06, predicted 0.057, estimated 0.057
shirt      true 0.50, predicted 0.303, estimated 0.503
sneaker    true 0.06, predicted 0.060, estimated 0.057
bag        true 0.06, predicted 0.066, estimated 0.051
ankle boot true 0.06, predicted 0.053, estimated 0.054
```

The classifier predicts shirts for well under half of the target images, between about a quarter and two fifths of them in our runs: shirts are the class it recognizes least often, and their errors spread over the other tops. Solving $\mathbf{C}\,\mathbf{p}_T = \mu$ undoes most of this mixing. The estimated fraction of shirts comes out near 0.5, and the estimates of the other classes lie within a few hundredths of the truth.

With the estimate we can adapt the classifier to the target without retraining it. By Bayes' theorem, multiplying the predicted probability of each class by $p_T(y)/p_S(y)$ and renormalizing turns probabilities for the source into probabilities for the target, provided the former are calibrated, which Section 3.5 found nearly true for this model. Multiplying a probability by a weight adds the logarithm of the weight to the logit, and the renormalization does not change which class is largest.

```
log_w = np.log(np.maximum(p_hat, 1e-6) / 0.1)  # log p_T(y) - log p_S(y)
accuracy = lambda O: (O.argmax(axis=1) == y[tgt]).mean()
print(f'target accuracy: {accuracy(O[tgt]):.3f} as trained, '
      f'{accuracy(O[tgt] + log_w):.3f} adapted to the estimated proportions')
```

```
target accuracy: 0.720 as trained, 0.820 adapted to the estimated proportions
```

The adapted classifier predicts shirts more readily, and its accuracy on the target rises by several points, without a single target label.

The estimate needs the label shift assumption, no target class that is absent from the source, and an invertible $\mathbf{C}$. Invertibility holds when the classifier gets the majority of each class right, $C_{jj} > \frac{1}{2}$ for every j (exercise 4); when $\mathbf{C}$ is nearly singular, small sampling errors in

μ and in the estimated $\mathbf{C}$ become large errors in $\mathbf{p}_T$. For a rare class the solution can even come out negative. In the screening example the estimate is $(\mu_1 - 0.05)/0.85$, where μ_1 is the fraction predicted sick. At a prevalence of 1% we expect $\mu_1 \approx 0.0585$, with a standard error of about 0.0074 on 1,000 target inputs, so μ_1 falls below 0.05, and the estimate below 0, in about one sample in eight. In practice one sets negative estimates to 0 and rescales the others to sum to 1, or solves a least-squares problem restricted to proportions. Adjusting a classifier to new class proportions goes back at least to Saerens et al. (2002); Lipton et al. (2018) showed that the confusion-matrix estimate is consistent for any fixed classifier with invertible $\mathbf{C}$.

Classes are not the only groups whose proportions can change. Under *subpopulation shift* it is groups such as hospitals, or birds photographed on water and on land, that are mixed differently in training and in deployment, and a classifier that leans on the background behind a bird can be accurate on average and fail on the few birds whose background and label disagree (Sagawa et al. 2020). When the groups are known, weighting the training examples by group, as we weighted them by class, is the first remedy, and benchmarks of such shifts report the lowest accuracy over the groups rather than the average (Koh et al. 2021).

3.7.4 Concept Shift and Feedback

Two cases remain, and reweighting helps in neither: the labeling rule itself may change, and the model's own decisions may change the data.

3.7.4.1 Updating with Fresh Labels

Concept shift changes $p(y \mid \mathbf{x})$, the very relation that the model estimates. Reweighting old examples cannot repair it, because their labels are no longer right. Only new labeled data help. When the change is gradual, as with news whose topics change from week to week, a common response is to keep the current parameters and continue training on recent labeled examples. When the change is abrupt, as when new diagnostic criteria take effect, examples labeled under the old definition describe a different task, and the model must be retrained on data labeled under the new one. Either way, we notice the change only by monitoring the model's accuracy on freshly labeled examples.

3.7.4.2 Decisions That Change the Data

Deploying a model turns its predictions into decisions, and decisions can change the data that the model sees next. The footwear model of the opening changed how applicants dressed. In predictive policing the effect acts on the labels (Figure 3.12). Patrols are sent where the model forecasts crime, more patrols record more of the crimes committed there, and the retrained model sends still more patrols. Recorded crime measures where the police looked as well as where crimes happened, so the loop can widen a small initial difference between two neighborhoods with the same crime rate.

None of the corrections in this section applies here: reweighting assumes a target distribution that exists independently of the model, while here the model helps produce it. Detecting the loop requires data that the model's decisions did not select, for instance from patrols sent to randomly chosen neighborhoods. Beyond detection lie questions that statistics cannot settle: whether the decision system serves an appropriate goal, and whether its errors

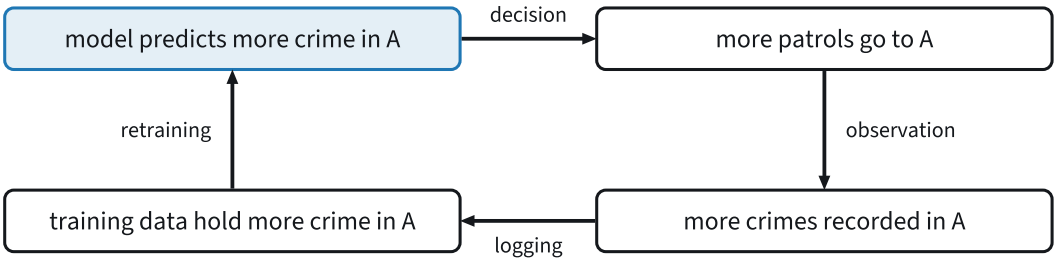


Figure 3.12: A feedback loop in predictive policing. Predictions decide where patrols go, patrols decide which crimes are recorded, and the records become the next training data.

fall unequally on different groups. A medical classifier must be judged by the costs of its errors in each population it serves, not by its average accuracy. Tuning its thresholds to those costs does not by itself make it fair: when a disease is more common in one group than in another, an imperfect test cannot have the same false positive rate, false negative rate, and precision in both. Problems in which a learner's actions determine what it observes next are the subject of reinforcement learning (Chapter 14).

3.7.5 Summary

A test set predicts performance in deployment only when deployment data come from the same distribution, and nothing transfers without an assumption about what stays fixed. Under covariate shift the labeling rule stays fixed, and weighting each training example by $p_T(\mathbf{x})/p_S(\mathbf{x})$, estimated by a classifier that tells target inputs from source inputs, corrects training; in our example it raised target accuracy from about half to about 95%. It requires the source to cover the target and loses effective sample size as the weights spread. Under label shift each class keeps its appearance, and the weights $p_T(y)/p_S(y)$ come from solving $\mathbf{C}\,\mathbf{p}_T = \mu$. Concept shift requires fresh labels, and when a model's decisions change its own data, statistical correction must be complemented by an analysis of the decisions.

3.7.6 Exercises

1. **Feedback loops.** A search engine changes how it ranks results. It then retrains its ranker on the clicks logged under the new ranking, with a query-document pair as the input $\mathbf{x}$ and a click as the label y.
    1. Describe one way users and one way advertisers might respond to the new ranking.
    2. For each response, state whether it changes $p(\mathbf{x})$, $p(y \mid \mathbf{x})$, or both in the next training set. Then decide whether the response is an instance of Goodhart's law.
2. **Missing support.** Suppose that $p_S(\mathbf{x}) = 0$ on a region $\mathcal{A}$ of inputs that has probability $\pi > 0$ under the target, and that $p_S(\mathbf{x}) > 0$ everywhere else. Define $\beta(\mathbf{x}) = p_T(\mathbf{x})/p_S(\mathbf{x})$ outside $\mathcal{A}$.
    1. Show that the weighted source risk $\mathbb{E}_{(\mathbf{x},y)\sim p_S}[\beta(\mathbf{x})\, l(f(\mathbf{x}), y)]$ equals the target risk with the contribution of $\mathcal{A}$ removed. Write the missing term as π times a conditional expectation.

2. Explain why no choice of weights on source examples can recover the missing term. Construct two models that agree outside $\mathcal{A}$ and have the same weighted source risk but different target risks.

3. code **When reweighting fails.** The code of Section 3.7.2 (`fit_logreg`, `X_src`, `X_tgt`, `label`) places the target mean at $(2, 0)$. Sweep the target mean over $(s, 0)$ for $s \in \{0, 0.5, 1, 2, 3, 4, 6\}$, keeping $n = 1000$ and the labeling rule fixed.
   1. For each s, report the accuracy of the domain classifier h on held-out pooled data, and interpret its growth with s in terms of how detectable the shift is.
   2. For each s, record the target accuracy of the unweighted, weighted, and clipped ($c = 5$) models, and of always predicting the most common target class. Explain why target accuracy alone cannot show whether reweighting worked at large s.

4. **The label shift system.** Let $\mathbf{C}$ be the column-normalized confusion matrix of Equation 3.42.
   1. Show that $\mathbf{C}$ is invertible whenever $C_{jj} > \frac{1}{2}$ for every class j. Hint: suppose that $\mathbf{v}^\top \mathbf{C} = \mathbf{0}$ for some $\mathbf{v} \neq \mathbf{0}$, look at the column j for which $|v_j|$ is largest, and use that the column sums to 1.
   2. Give an example of a classifier for which $\mathbf{C}$ is singular, and describe what μ can still reveal about $\mathbf{p}_T$ in that case.
   3. For two classes, show that an error δ in the fraction μ_1 of positive predictions changes the estimated fraction of positives by $\delta/(C_{11} - C_{10})$. Using the standard error $\sqrt{\mu_1(1-\mu_1)/m}$ of a fraction estimated from m target inputs, find how many target inputs the screening example needs for a standard error of 0.01 in the estimated fraction of sick patients. Repeat for a weaker test with $C_{11} = 0.6$ and $C_{10} = 0.4$ on the same population.

5. **Features missing at serving time.** A model forecasts a store's daily revenue at the start of each day. Its training records hold each past day's final customer count as a feature, and the model performs well in offline evaluation on held-out records. At serving time the pipeline fills this feature with the number of customers counted so far that day.
   1. Compare the distribution of this feature, and its relation to revenue, in the training records and at serving time. State which of the section's shift assumptions, if any, holds between the two.
   2. Explain why the offline evaluation could not reveal this mismatch, and why the reweighting of Section 3.7.2 cannot repair it.
   3. Propose a rule for selecting training features that avoids this failure.

   Adapted from Google's Machine Learning Crash Course, "Monitoring pipelines".

6. code **Learned versus true weights.** ⊠ Reuse the sweep of exercise 3 over target means $(s, 0)$.
   1. Derive the true log-ratio $\log(p_T(\mathbf{x})/p_S(\mathbf{x}))$ as a function of s, and compare it with the learned h at each s. Explain how the domain classifier can reach high accuracy at large s while h is far from the true log-ratio.
   2. Show that under the source the true weights satisfy $\mathbb{E}[\beta(\mathbf{x})] = 1$ and $\mathbb{E}[\beta(\mathbf{x})^2] = e^{s^2}$, so that for large n their effective sample size is about ne^{-s^2}. Evaluate this for $n = 1000$ and $s = 1, 2, 3$.
   3. For each s, compute the effective sample size of the true weights and of the learned weights on the actual source sample. Explain why at $s = 2$ the sam-

ple value is usually much larger than $1000\,e^{-4} \approx 18$, and what this means for judging the reliability of a reweighted fit from the sample alone. Does clipping at $c = 5$ repair the shortage?

7. **Shift assumptions from unlabeled data.** ⊠ In a binary source domain, $p_S(y = 1) = p_S(y = 0) = \frac{1}{2}$, and the scalar input satisfies $x \mid y = 1 \sim \mathcal{N}(1, 1)$ and $x \mid y = 0 \sim \mathcal{N}(-1, 1)$. Hence $p_S(y = 1 \mid x) = 1/(1 + \exp(-2x))$. Unlabeled target inputs follow the mixture

$$p_T(x) = 0.8\,\mathcal{N}(x; 1, 1) + 0.2\,\mathcal{N}(x; -1, 1). \tag{3.43}$$

   1. Assume label shift. Give $p_T(y = 1)$ and the threshold on x of the Bayes-optimal target classifier.
   2. Assume instead covariate shift with the same $p_T(x)$. Compute $p_T(y = 1)$ numerically and give the Bayes-optimal threshold.
   3. Use your results to explain why no statistic of the unlabeled target sample can decide which of the two corrections of this section applies.

Chapter 4

Multilayer Perceptron

The softmax regression of Chapter 3 weighs each of the 784 pixels of a clothing image and adds up the evidence for each of the ten classes. A pixel therefore adds the same evidence for a class whatever the pixels around it show. Yet a bright pixel may belong to a sleeve, a strap, or the background, and only its neighbors can tell which. How do we build a model in which the evidence from one input depends on the others, and how do we train it?

The trouble starts with a much smaller example than a sleeve: no threshold on a weighted sum computes the exclusive or of two bits. Marvin Minsky and Seymour Papert made the point in 1969 about the *perceptron*, a classifier proposed by Frank Rosenblatt in 1958 that learns the weights of a weighted sum from examples. Feeding the classifier new features computed from the inputs, such as the product of the two bits, escapes the limit. But somebody has to supply the features, and for years no widely used method learned them. A *multilayer perceptron* (MLP) computes them in hidden layers, each an affine map followed by a fixed nonlinear function. In 1986, Rumelhart, Hinton, and Williams showed that backpropagation trains all the layers together by gradient descent, so that the hidden layers learn useful features (Rumelhart et al. 1986).

The loss, the optimizer, and the training loop of softmax regression carry over unchanged, but hidden layers raise three new problems. Each training step needs derivatives with respect to hundreds of thousands of parameters. A deep network whose initial weights have the wrong scale does not train at all. And a network with more parameters than training examples could memorize its data, yet usually does not. The first two problems have satisfactory answers. The third is the deepest, and what we can say about it is still incomplete.

We begin by building the model: Section 4.1 shows which functions a network with hidden layers can represent and why it needs a nonlinearity, and Section 4.2 trains one on the clothing images to see what it gains over softmax regression, where, and at what cost. The three problems then get a section each, in that order (Section 4.3, Section 4.4, Section 4.5). Dropout (Section 4.6) is a regularizer made for hidden units, and we try it where a network memorizes wrong labels. To finish, Section 4.7 takes the whole toolkit away from images, to a Kaggle competition on house prices.

The chapter assumes what Section 2.6 and Section 2.7 said about overfitting, and it treats fully connected networks only. Layers that exploit the structure of images and sequences, and the optimizers and normalization layers that train very deep networks, follow in later chapters.

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.

Books and notes

- Deep Learning — Goodfellow, Bengio & Courville — free HTML; Chapter 6 covers hidden units, activation functions, universal approximation, and backpropagation (Section 4.1, Section 4.3), and Chapter 7 covers dropout (Section 4.6).
- Neural Networks and Deep Learning, Chapter 2 — Michael Nielsen — free online; derives backpropagation step by step from the chain rule (Section 4.3).
- Understanding Deep Learning — Simon J. D. Prince — free PDF with notebooks; its chapters on shallow and deep networks, gradients and initialization, and generalization accompany Section 4.1, Section 4.4, and Section 4.5.
- Matrix calculus and automatic differentiation in this book's mathematics appendix (Section 34.4 and Section 34.5) — reverse-mode differentiation with proofs, the background of Section 4.3.
- Concentration and generalization in the mathematics appendix (Section 36.10) — uniform convergence, the minimum-norm mechanism behind double descent, and a numerical double-descent example with random features (Section 4.5).

Courses and tutorials

- CS231n notes: Neural Networks, Parts 1 and 2 — Stanford — free; activation functions, weight initialization, and dropout in practice (Section 4.1, Section 4.4, Section 4.6); the companion note Backpropagation, Intuitions works through gradients on small graphs (Section 4.3).
- micrograd — Andrej Karpathy — free; a scalar automatic-differentiation engine in about 100 lines, with a video walkthrough that builds it from scratch (Section 4.3).
- TensorFlow Playground — Smilkov & Carter — free, in the browser; vary depth, width, and activation and watch a decision boundary form (Section 4.1).
- Double Descent — MLU-Explain — free, interactive; animates the second descent past the interpolation threshold (Section 4.5).

Papers

- Approximation by Superpositions of a Sigmoidal Function — Cybenko (1989) — the universal approximation theorem for one hidden layer of sigmoid units (Section 4.1; paywalled).
- Learning Representations by Back-propagating Errors — Rumelhart, Hinton & Williams (1986) — the paper that made backpropagation and learned hidden representations widely known (Section 4.3; paywalled).
- Understanding the Difficulty of Training Deep Feedforward Networks — Glorot & Bengio (2010) and Delving Deep into Rectifiers — He, Zhang, Ren & Sun (2015) — derive Xavier and He initialization from keeping the variance of signals constant across layers (Section 4.4).
- Understanding Deep Learning Requires Rethinking Generalization — Zhang, Bengio, Hardt, Recht & Vinyals (2017) — networks that fit random labels, the experiment that opens Section 4.5.
- Reconciling Modern Machine-Learning Practice and the Bias–Variance Trade-off — Belkin, Hsu, Ma & Mandal (2019) — the double-descent curve (Section 4.5).

- Dropout: A Simple Way to Prevent Neural Networks from Overfitting — Srivastava et al. (2014) — the dropout paper (Section 4.6).
- Ames, Iowa: Alternative to the Boston Housing Data — De Cock (2011) — the dataset behind the house-price competition, and how it was assembled (Section 4.7).

4.1 Multilayer Perceptrons

The softmax regression of Chapter 3 adds up evidence pixel by pixel, so each input moves a class score by a fixed amount, independently of all the others. What exactly does this rule out, and what must a model have beyond a weighted sum to escape it?

It is instructive to start with the smallest problem on which a weighted sum fails. It has four points. Label each corner of the unit square by whether its two coordinates differ, the *exclusive or* (XOR) of two bits. No line separates the two classes, so every linear classifier gets at least one of the four points wrong.

A *multilayer perceptron* (MLP) removes this limit by computing new features before it classifies. A hidden layer applies an affine map followed by a fixed nonlinear function, and an output layer, which is softmax regression, classifies the result. Both layers are trained together, with the loss and the minibatch stochastic gradient descent (SGD) of the previous chapter. This section defines the model and builds a network with two hidden units that computes XOR. It then asks which functions such networks can represent and compares the nonlinear functions in common use.

4.1.1 Limits of Linear Models

4.1.1.1 Monotonicity and Interactions

To see what a linear model cannot express, we look again at how it scores a class. In softmax regression (Section 3.1), an input $\mathbf{x} \in \mathbb{R}^d$ gives class j the score

$$o_j = \sum_{i=1}^{d} x_i w_{ij} + b_j. \tag{4.1}$$

Each score is therefore *monotonic* in each input: it rises with x_i if $w_{ij} > 0$ and falls if $w_{ij} < 0$. Moreover, the inputs do not interact: raising x_i by one changes o_j by w_{ij}, no matter what the other inputs are. Both properties often fail. The risk of illness rises as body temperature departs from 37°C in either direction. This case we can repair by hand, using the distance from 37°C as the input. Interactions are the more serious limit, and they occupy the rest of this section. Images resist repairs by hand: whether a bright pixel is evidence for a shirt or for a bag depends on its neighbors, and writing down features that capture all such dependencies is hopeless. We want the model to learn its features from data instead.

4.1.1.2 The XOR Problem

The smallest problem in which inputs interact has two binary inputs and the label

x_1	x_2	y
0	0	0

x_1	x_2	y
0	1	1
1	0	1
1	1	0

which is 1 if exactly one input is 1. Changing x_1 from 0 to 1 changes the label from 0 to 1 when $x_2 = 0$, but from 1 to 0 when $x_2 = 1$. With two classes, softmax regression predicts class 1 where the difference of the two logits is positive (Section 3.1). This difference is an affine score $o = w_1 x_1 + w_2 x_2 + b$, so the decision boundary is the line $o = 0$.

Proposition. For all w_1, w_2, b, the rule "predict class 1 if $w_1 x_1 + w_2 x_2 + b > 0$" misclassifies at least one of the four points of XOR.

Proof. Suppose that some w_1, w_2, b classify all four points correctly. The two points of class 1 give $w_2 + b > 0$ and $w_1 + b > 0$, and adding the two inequalities gives $w_1 + w_2 + 2b > 0$. The two points of class 0 give $b \leq 0$ and $w_1 + w_2 + b \leq 0$, and adding these gives $w_1 + w_2 + 2b \leq 0$, a contradiction. ■

The proof can also be seen geometrically. Both pairs of points have the midpoint $(1/2, 1/2)$, where an affine score equals the average of its values at either pair, so it would have to be positive and not positive at once. Three out of four is the best a line can do: the line $x_1 + x_2 = 3/2$, for example, misclassifies only $(0, 0)$.

4.1.2 Hidden Layers

4.1.2.1 The Model

An MLP first computes h new features, each an affine function of the input followed by a fixed nonlinear function ϕ, and then applies a linear model to them. The features form a *hidden layer* of *hidden units*, so called because their values are neither given by the data nor reported as outputs. Figure 4.1 shows an MLP with four inputs, five hidden units, and three outputs. Every input feeds every hidden unit, and every hidden unit feeds every output, so both layers are fully connected. The inputs involve no computation, so we count only the layers with parameters: this MLP has two layers, one of them hidden. The name comes from the perceptron, an early classifier that thresholds a weighted sum. Each hidden unit resembles one, with ϕ in place of the threshold, and an MLP arranges such units in layers.

We keep the layout of Section 3.1.1.4: the n examples of a minibatch are the rows of $\mathbf{X} \in \mathbb{R}^{n \times d}$, and each weight matrix has one row per input and one column per output of its layer. With a superscript numbering the layers, the hidden layer has weights $\mathbf{W}^{(1)} \in \mathbb{R}^{d \times h}$ and biases $\mathbf{b}^{(1)} \in \mathbb{R}^h$, the output layer has weights $\mathbf{W}^{(2)} \in \mathbb{R}^{h \times q}$ and biases $\mathbf{b}^{(2)} \in \mathbb{R}^q$, and the MLP computes

$$\begin{aligned} \mathbf{H} &= \phi\left(\mathbf{X}\mathbf{W}^{(1)} + \mathbf{b}^{(1)}\right), \\ \mathbf{O} &= \mathbf{H}\mathbf{W}^{(2)} + \mathbf{b}^{(2)}. \end{aligned} \tag{4.2}$$

The biases are added to every row by broadcasting, and ϕ acts on each entry separately. Row i of $\mathbf{H} \in \mathbb{R}^{n \times h}$ is the *hidden representation* of example i, and row i of $\mathbf{O} \in \mathbb{R}^{n \times q}$ holds its logits, which the softmax turns into probabilities as before. For a single example $\mathbf{x}$, the

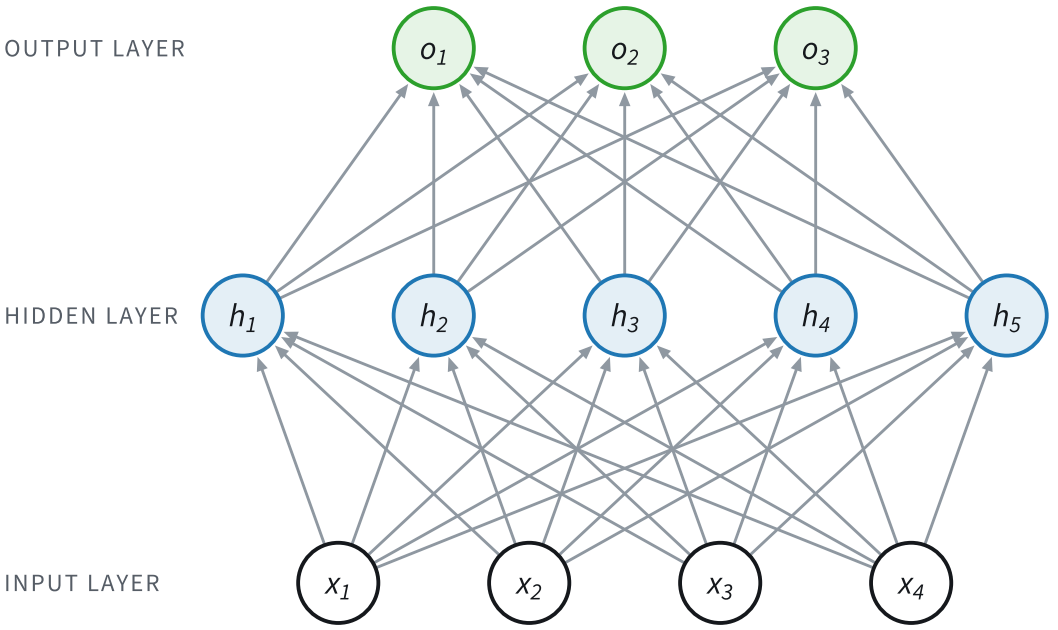


Figure 4.1: An MLP with four inputs, a hidden layer of five units, and three outputs. Both layers are fully connected.

model reads $\mathbf{h} = \phi(\mathbf{W}^{(1)\top}\mathbf{x} + \mathbf{b}^{(1)})$ and $\mathbf{o} = \mathbf{W}^{(2)\top}\mathbf{h} + \mathbf{b}^{(2)}$. The transposes appear only because $\mathbf{x}$ and $\mathbf{h}$ are columns, while examples are rows of $\mathbf{X}$.

The function ϕ is the *activation function*. Its inputs, the entries of $\mathbf{X}\mathbf{W}^{(1)} + \mathbf{b}^{(1)}$, are called *pre-activations*, and its outputs *activations*. Minibatch SGD fits all four parameter arrays together on the cross-entropy loss, with gradients from automatic differentiation, so the network learns a representation and a linear classifier of that representation at once.

The hidden layer adds parameters, and rather a lot of them. With d inputs, h hidden units, and q outputs, an MLP has $(d+1)h + (h+1)q$ parameters, against $(d+1)q$ for softmax regression. For Fashion-MNIST, with $d = 784$ and $q = 10$, a hidden layer of $h = 256$ units gives $785 \cdot 256 + 257 \cdot 10 = 203{,}530$ parameters, 26 times the 7,850 of softmax regression.

4.1.2.2 The Role of the Nonlinearity

Why apply ϕ at all? Suppose that we omit it, so that $\mathbf{H} = \mathbf{X}\mathbf{W}^{(1)} + \mathbf{b}^{(1)}$. Substituting $\mathbf{H}$ into the output layer gives

$$\mathbf{O} = \left(\mathbf{X}\mathbf{W}^{(1)} + \mathbf{b}^{(1)}\right)\mathbf{W}^{(2)} + \mathbf{b}^{(2)} = \mathbf{X}\mathbf{W} + \mathbf{b},$$

where $\mathbf{W} = \mathbf{W}^{(1)}\mathbf{W}^{(2)}$ and $\mathbf{b} = \mathbf{W}^{(2)\top}\mathbf{b}^{(1)} + \mathbf{b}^{(2)}$. The two layers compute an affine function of the input, the very one that softmax regression computes with the weights $\mathbf{W}$ and the bias $\mathbf{b}$. The hidden layer adds parameters but no new functions (it can even remove some, Exercise 1), and the network still fails on XOR.

With a nonlinear ϕ, the substitution no longer simplifies. The simplest useful choice, and the default in this book, is the *rectified linear unit* (ReLU),

$$\mathrm{ReLU}(z) = \max(z, 0), \tag{4.3}$$

which keeps positive values and replaces negative ones by zero.

4.1.2.3 Solving XOR

How many hidden units does XOR need? Two ReLU units suffice. One choice of weights (Goodfellow et al. 2016) is

$$\mathbf{W}^{(1)} = \begin{pmatrix} 1 & 1 \\ 1 & 1 \end{pmatrix}, \quad \mathbf{b}^{(1)} = (0, -1), \quad \mathbf{w}^{(2)} = (1, -2), \quad b^{(2)} = 0, \tag{4.4}$$

where the output layer has a single output o, so its weights form one column $\mathbf{w}^{(2)}$. Both hidden units see the sum $s = x_1 + x_2$. The first, $h_1 = \mathrm{ReLU}(s) = s$, counts the inputs that are 1. The second, $h_2 = \mathrm{ReLU}(s - 1)$, is positive only when both inputs are 1, and the output $o = h_1 - 2h_2$ subtracts 2 in that case. We check the network on the four inputs, stored as the rows of $\mathbf{X}$.

```
X = np.array([[0, 0], [0, 1], [1, 0], [1, 1]])
W1, b1 = np.array([[1, 1], [1, 1]]), np.array([0, -1])
w2, b2 = np.array([1, -2]), 0
H = np.maximum(X @ W1 + b1, 0)  # hidden layer with ReLU
o = H @ w2 + b2  # output layer
for x, h, out in zip(X, H, o):
    print(f'x = {x}, h = {h}, o = {out}')
```

```
x = [0 0], h = [0 0], o = 0
x = [0 1], h = [1 0], o = 1
x = [1 0], h = [1 0], o = 1
x = [1 1], h = [2 1], o = 0
```

The output equals the XOR label on every input. As a function of s, the output $o = \mathrm{ReLU}(s) - 2\,\mathrm{ReLU}(s - 1)$ rises from 0 at $s = 0$ to 1 at $s = 1$ and falls back to 0 at $s = 2$. The second unit *folds* the line of values of s at $s = 1$, so that its two ends reach the same output. In the hidden space (Figure 4.2, right), the two inputs of class 1 land on the same point $(1, 0)$, and the line $h_1 - 2h_2 = 1/2$ separates the classes. In the input space (left), the network predicts class 1 in the band $1/2 < x_1 + x_2 < 3/2$, a region that no single line can bound.

The hidden layer has re-represented the inputs so that a linear classifier suffices. Here we chose its weights by hand. For a problem such as Fashion-MNIST nobody could, and training has to choose them instead.

4.1.2.4 Deeper Networks

Nothing stops us from stacking more hidden layers. Starting from $\mathbf{H}^{(0)} = \mathbf{X}$, hidden layer k computes

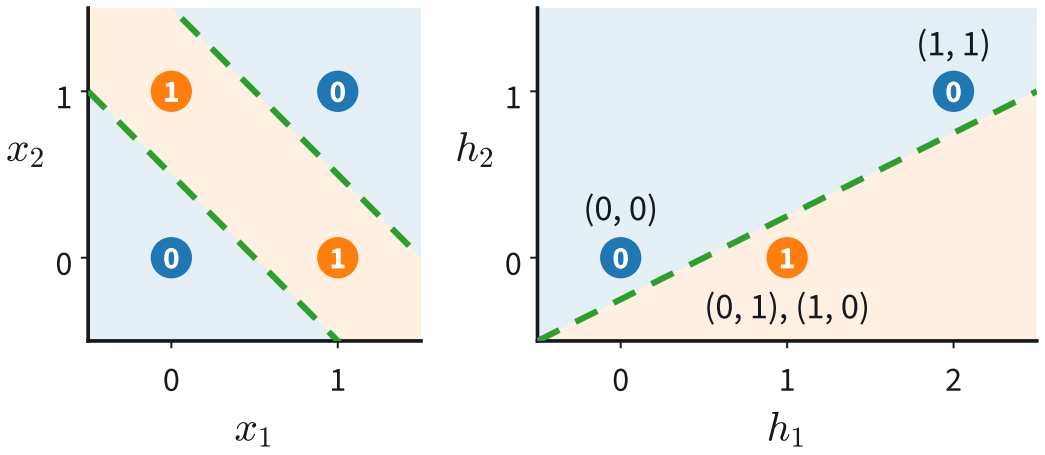


Figure 4.2: XOR and the two-unit ReLU network that computes it. Left: the four inputs, labeled by XOR (blue: class 0, orange: class 1); the network predicts class 1 in the shaded band between the dashed lines $x_1 + x_2 = 1/2$ and $x_1 + x_2 = 3/2$. Right: the same inputs after the hidden layer $\mathbf{h} = \text{ReLU}(\mathbf{W}^{(1)\top}\mathbf{x} + \mathbf{b}^{(1)})$, each marked with the input it came from; the two inputs of class 1 land on the same point $(1, 0)$, and the dashed line $h_1 - 2h_2 = 1/2$ separates the classes.

$$\mathbf{H}^{(k)} = \phi\left(\mathbf{H}^{(k-1)}\mathbf{W}^{(k)} + \mathbf{b}^{(k)}\right), \quad k = 1, \ldots, K-1, \tag{4.5}$$

and the output layer computes $\mathbf{O} = \mathbf{H}^{(K-1)}\mathbf{W}^{(K)} + \mathbf{b}^{(K)}$. Such a network has K layers, $K-1$ of them hidden, and each hidden layer may have its own width. Each hidden layer computes a new representation from the one below it, and the output layer classifies the last one. For $K = 2$ this is Equation 4.2. Whether several narrow layers do better than one wide layer is a question we take up once we know what a single hidden layer can do.

4.1.3 Expressive Power

4.1.3.1 Universal Approximation

Two hidden units sufficed for XOR. Can a network with one hidden layer approximate every function that we might want to learn? For a single input x and ReLU units, a picture answers the question. A hidden unit with input weight 1, bias $-t$, and output weight a adds $a\,\text{ReLU}(x - t)$ to the output. This term is a *hinge*: zero to the left of its joint t and a line of slope a to the right of it. A sum of h hinges and a constant is continuous and linear between the joints, so it has at most $h + 1$ linear pieces (Figure 4.3). Conversely, every polyline on a closed interval is such a sum: one hinge at the left end with the slope of the first piece, and one hinge at each later joint with the change of slope there.

The approximation result for one input follows. A continuous function f on a closed interval changes by less than any given $\epsilon > 0$ between points that are close enough together. The polyline through the values of f at such closely spaced points therefore stays within ϵ of f on the whole interval, and a hidden layer with one unit per joint computes it.

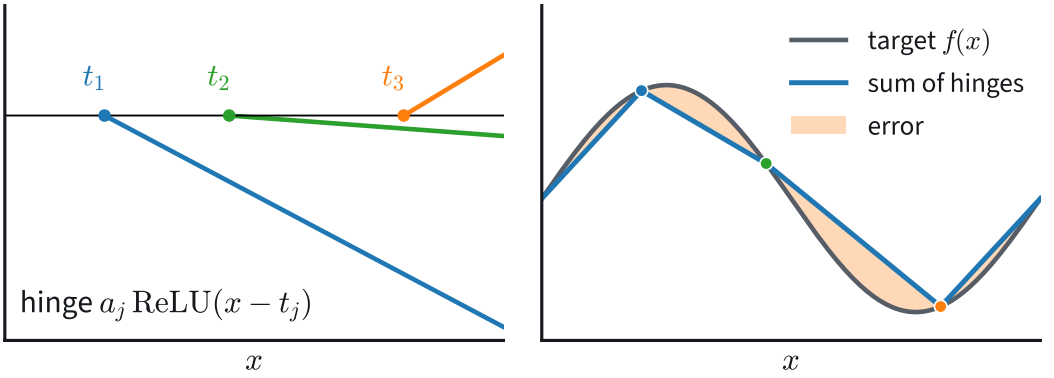


Figure 4.3: Approximation by hinges. Left: three hidden units, each contributing a hinge $a_j \, \mathrm{ReLU}(x - t_j)$ whose joint t_j is marked. Right: added to a constant and to one more hinge whose joint lies at the left end of the interval, which gives the first piece its slope, the three hinges form a polyline with 4 pieces (blue) that follows a smooth target (gray); the shaded area is the error, which shrinks as more joints are added.

Neither the single input nor the ReLU is essential. The *universal approximation theorem* states that the same conclusion holds for any number of inputs, on any closed and bounded set, and for every continuous activation function that is not a polynomial (Cybenko (1989) for sigmoids, Leshno et al. (1993) in general). Polynomials must be excluded, because with a polynomial activation function the network computes a polynomial of bounded degree, and such polynomials cannot approximate every continuous function. The general proof is harder than our picture, and we skip it. Exercise 6 proves the theorem for two inputs.

4.1.3.2 Limits of the Theorem

The theorem says that good weights exist, and it leaves three questions open. The first is width. The construction above uses one unit per joint, and a target that changes direction often needs many joints. With many inputs, constructions of this kind cover the input space with small cells, and the number of cells, and with it the number of hidden units, can grow exponentially with the number of inputs. The second question is training: the theorem does not say that minibatch SGD, started from random weights, finds good weights. The third is generalization. Approximating a function that we know everywhere differs from learning it from n examples, and a network flexible enough to approximate any function can also fit the noise in its training labels.

4.1.3.3 Depth

Width adds linear pieces one unit at a time: on one input, h hidden units give at most $h + 1$ pieces. Depth can multiply them. Consider the tent

$$g(z) = 2 \, \mathrm{ReLU}(z) - 4 \, \mathrm{ReLU}(z - 1/2), \tag{4.6}$$

which rises from 0 to 1 on $[0, 1/2]$ and falls back to 0 on $[1/2, 1]$. It is the XOR network read along the diagonal: with $s = 2z$, the output $\mathrm{ReLU}(s) - 2 \, \mathrm{ReLU}(s - 1)$ equals $g(z)$. Now feed g its own output. As z runs from 0 to 1/2, $g(z)$ runs from 0 to 1, so $g(g(z))$ traces the whole

tent. As z runs on from 1/2 to 1, $g(z)$ runs back from 1 to 0, and $g(g(z))$ traces the tent again in reverse. The first layer folds the interval $[0, 1]$ in half, and everything after it is computed on both halves. Each composition with g doubles the number of pieces (Figure 4.4). How many units does it take? Each copy of g is a hidden layer of two ReLU units followed by a weighted sum, and the next copy begins with an affine map. Two affine maps in a row are one, as we saw for networks without activation functions, so each composition adds one hidden layer of two units and nothing else.

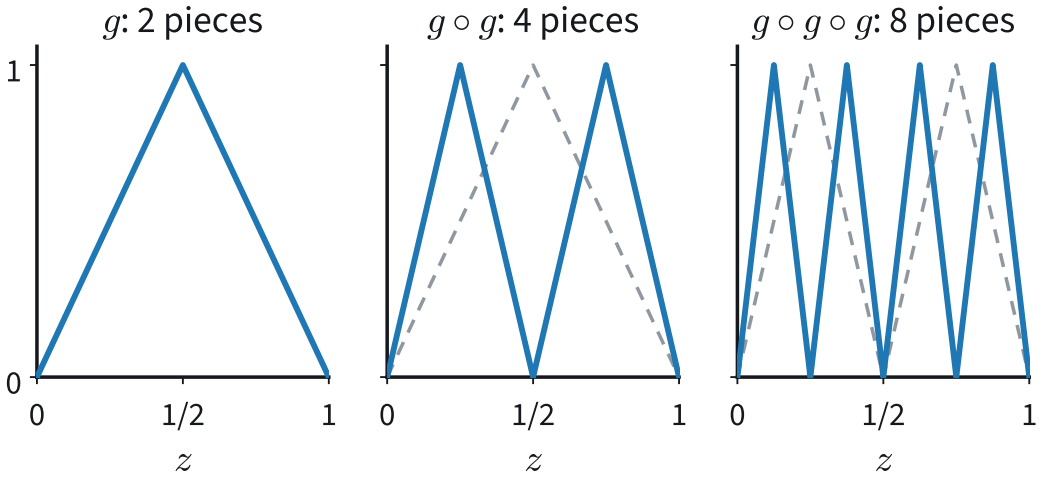


Figure 4.4: Depth folds the input. The tent g, computed by one hidden layer of two ReLU units, and its compositions $g \circ g$ and $g \circ g \circ g$, computed by two and three such layers. Each composition splits every piece of the previous function (dashed) in two.

Thus composing g with itself m times gives a zigzag with 2^m pieces, computed by m hidden layers of two units, $2m$ hidden units in all. A single hidden layer needs at least $2^m - 1$ units for the same function, by the count of pieces above. For $m = 10$, this means 20 units in ten hidden layers against at least 1,023 in one. Telgarsky (2016) used this construction to prove that some functions computed by deep, narrow networks require exponentially wide shallow networks. The zigzag is a rather special function, of course. Depth does not help with every function, and nothing guarantees that training finds such compact solutions when they exist.

4.1.4 Activation Functions

So far almost any ϕ would have served. Every nonlinear one prevents the collapse into a single affine map, and every continuous one that is not a polynomial makes one hidden layer a universal approximator. The choice begins to matter when the network is trained. Gradient descent changes each parameter in proportion to the derivative of the loss l with respect to it. For one example, hidden unit j computes $h_j = \phi(z_j)$ from its pre-activation $z_j = \sum_i x_i w_{ij}^{(1)} + b_j^{(1)}$, where $w_{ij}^{(1)}$ is an entry of $\mathbf{W}^{(1)}$. By the chain rule,

$$\frac{\partial l}{\partial w_{ij}^{(1)}} = \frac{\partial l}{\partial h_j}\, \phi'(z_j)\, x_i. \tag{4.7}$$

The gradient of every weight into unit j contains the same factor $\phi'(z_j)$. Where this factor is close to zero, none of these weights moves, however wrong the prediction. Figure 4.5 shows four activation functions in common use, with their derivatives.

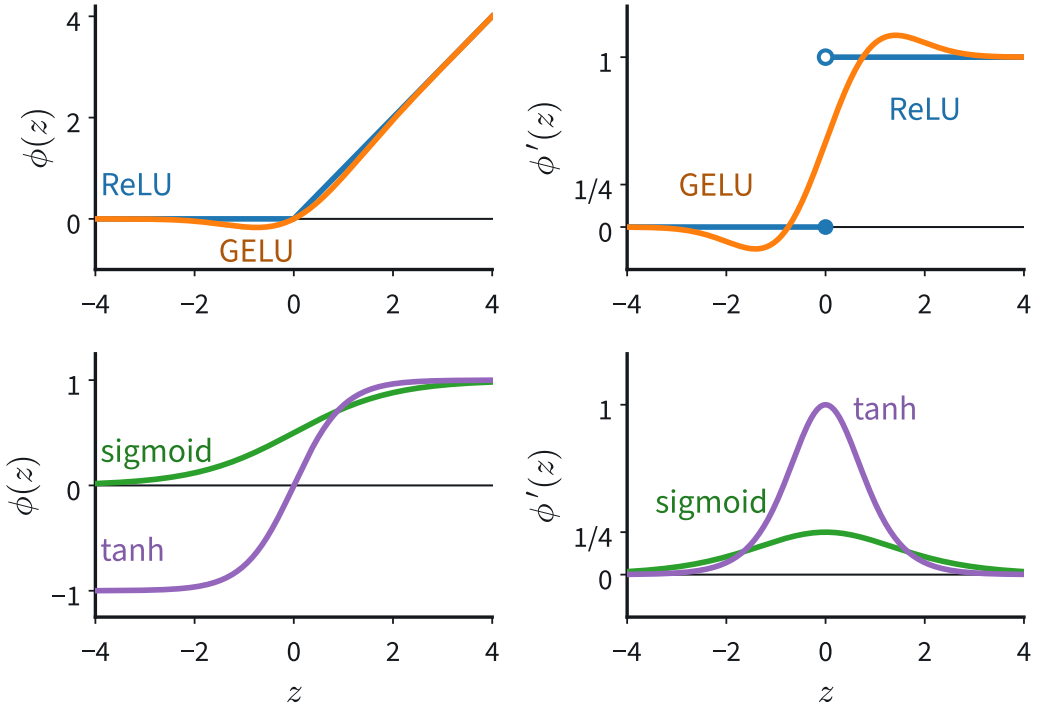


Figure 4.5: Four activation functions ϕ (left) and their derivatives ϕ' (right). Top: the ReLU and its smooth relative, the GELU. Bottom: the sigmoid and tanh, which squash their inputs into bounded intervals. The two derivative panels share one vertical scale: the derivative of the sigmoid never exceeds $1/4$, and the derivatives of the sigmoid and tanh vanish in both tails.

4.1.4.1 ReLU and GELU

The derivative of the ReLU is 1 for $z > 0$ and 0 for $z < 0$. At $z = 0$, where the ReLU has no derivative, the built-in ReLU uses 0, a choice that rarely matters because a pre-activation is seldom exactly zero. For every positive pre-activation, however large, the gradient passes through a ReLU unit unchanged. This is the main reason why the ReLU replaced the sigmoid as the default activation of hidden layers (Nair and Hinton 2010; Glorot et al. 2011). It is also cheap: a single comparison.

The flat side has a cost. A unit whose pre-activation is negative for every training example outputs zero, receives zero gradient, and never changes again. Such a unit is called *dead*. The leaky ReLU, $\max(z, \alpha z)$ with a small slope such as $\alpha = 0.01$, keeps a small gradient for negative inputs, so its units cannot die.

The *Gaussian error linear unit* (GELU) is a smooth relative of the ReLU,

$$\mathrm{GELU}(z) = z\,\Phi(z), \tag{4.8}$$

where Φ is the cumulative distribution function of the standard normal distribution. Since $\Phi(z)$ rises smoothly from 0 for very negative z to 1 for very positive z, the GELU matches the ReLU for $|z|$ beyond about 3 and rounds off its corner in between. Its slope at $z = 0$ is 1/2, and unlike the ReLU it dips below zero, to about -0.17 near $z = -0.75$. Many transformer models use the GELU (Hendrycks and Gimpel 2016). A close relative, the *SiLU* $z\,\sigma(z)$, replaces Φ by the sigmoid σ of the next subsection.

4.1.4.2 Sigmoid and Tanh

The sigmoid, $\sigma(z) = 1/(1 + \exp(-z))$, turns the logit of a two-class classifier into a probability (Section 3.1): it maps the real line onto the interval $(0, 1)$. Early neural networks used it in hidden layers as a smooth version of a threshold unit, which outputs 1 when its input exceeds a threshold and 0 otherwise (McCulloch and Pitts 1943). Unlike the threshold, the sigmoid has a derivative that gradient descent can use,

$$\sigma'(z) = \sigma(z)\,(1 - \sigma(z)). \tag{4.9}$$

It is largest at $z = 0$, where it equals $1/4$, and approaches zero in both tails. A unit whose pre-activation is large in magnitude is *saturated*: its output barely changes with its input, so it passes almost no gradient to the weights below it. At $z = \pm 4$, the factor is $\sigma(4)(1-\sigma(4)) \approx 0.018$.

Even at $z = 0$, every sigmoid unit between a weight and the loss shrinks the gradient by a factor of 4. Through a chain of ten sigmoid units with weights of 1, the gradient therefore shrinks by a factor of at least $4^{10} \approx 10^6$. Such *vanishing gradients* made deep sigmoid networks hard to train; Section 4.4 analyzes them and their remedies.

The *hyperbolic tangent*,

$$\tanh(z) = \frac{1 - \exp(-2z)}{1 + \exp(-2z)}, \tag{4.10}$$

maps the real line onto $(-1, 1)$. It is a sigmoid stretched to this interval and centered at zero (Exercise 3), and its derivative, $1 - \tanh^2(z)$, reaches 1 at $z = 0$, so an unsaturated tanh unit passes the whole gradient. In the tails it vanishes even faster than that of the sigmoid: at $z = \pm 4$ it is about 0.0013. Today the sigmoid serves mainly to produce probabilities and gates, numbers between 0 and 1 that decide what fraction of a signal passes, and tanh to keep the state of recurrent networks bounded.

We have not trained a single network in this section: every weight was chosen by hand. Whether training finds good weights, and whether they predict well on new data, are questions for experiment. The next section runs the first one, on Fashion-MNIST.

4.1.5 Summary

A linear classifier scores each class with a weighted sum of the inputs, so it cannot represent interactions between inputs, and the four points of XOR already defeat it. A multilayer perceptron first computes a hidden representation $\mathbf{H} = \phi(\mathbf{X}\mathbf{W}^{(1)} + \mathbf{b}^{(1)})$ and then classifies it

linearly, $\mathbf{O} = \mathbf{H}\mathbf{W}^{(2)} + \mathbf{b}^{(2)}$, training both layers together. Without the activation function ϕ, the layers collapse into one affine map; with it, two ReLU units fold the XOR inputs so that a line separates them. One hidden layer can approximate any continuous function, possibly only with very many units, and depth can produce exponentially many linear pieces from a linear number of units. None of this says that training finds such weights. Which activation function lets it try is decided by the derivative: the ReLU passes gradients of 0 or 1 and is the default for hidden layers, while the sigmoid and tanh saturate and pass at most 1/4 and 1 of the gradient.

4.1.6 Exercises

1. **Linear networks.** Without activation functions, stacking layers composes affine maps into another affine map. Consider such a network with input dimension d, output dimension q, and hidden widths $h_1, \dots, h_{K-1}$. Show that it represents exactly the affine maps $\mathbf{X}\mathbf{W} + \mathbf{b}$ whose weight matrix $\mathbf{W}$ has rank at most $\min(h_1, \dots, h_{K-1})$. Use this result to give an example in which adding a hidden layer to a linear model strictly shrinks its function class.

2. **XOR by hand.** Figure 4.2 shows a two-unit ReLU network that computes XOR by mapping the two inputs of class 1 to the same hidden point.

    1. Find weights for a two-unit ReLU network that also computes XOR but maps the two inputs of class 1 to *distinct* hidden points, and verify it on the four inputs.
    2. Show that no network with a single ReLU hidden unit and a linear output classifies XOR correctly, for any threshold applied to its output.

3. **Sigmoid and tanh.** The two squashing functions are closely related.

    1. Show that $\tanh(z) = 2\sigma(2z) - 1$.
    2. Show that for every MLP with tanh hidden activations there is an MLP of the same architecture with sigmoid hidden activations that computes the same function, and conversely. State how the weights and biases change.

4. code **Vanishing gradients.** Consider a chain of K sigmoid units of width one, $h_k = \sigma(z_k)$ with pre-activation $z_k = w_k h_{k-1} + b_k$. By the chain rule,

$$\frac{\partial h_K}{\partial h_0} = \prod_{k=1}^{K} w_k \sigma'(z_k). \tag{4.11}$$

    1. Explain why the weight feeding a *saturated* unit, one operating at large $|z_k|$, receives almost no gradient for any loss, and why a single saturated unit also stalls every earlier layer of the chain.
    2. Choose the biases so that every pre-activation equals 0, where σ' attains its maximum $1/4$. Compute the product for $K = 1, \dots, 20$, once with every $w_k = 1$ and once with every $w_k = 4$, and plot both curves on a logarithmic vertical axis.
    3. Repeat with every pre-activation equal to 5 and every $w_k = 1$. Find the weight w for which $w\sigma'(5) = 1$, and compare it with the weight that keeps the product constant at the origin. State what the comparison implies for training deep sigmoid networks in either regime.

5. **Depth and linear pieces.** ⊠ A one-hidden-layer ReLU network with h hidden units on the real line has at most $h + 1$ linear pieces, while each composition with the two-unit

tent of Figure 4.4 doubles the number of pieces. This problem makes the gain from depth exact for a network with three units per layer.

1. Take three hidden units $\mathrm{ReLU}(w_j x + b_j)$ with input weights $w_1 = w_2 = 1$ and $w_3 = -1$ and with joints $-b_j/w_j$ at $1/6$, $2/6$, and $4/6$. Choose output weights and an output bias so that the network's output f takes the values $0, 1, 0, 1, 0$ at $x = 0, 1/6, 2/6, 4/6, 1$.
2. Count the linear pieces of $f \circ f$ on $[0, 1]$, and of the m-fold composition $f \circ \cdots \circ f$. Explain, in terms of how composing with f folds the graph, why each composition multiplies the count rather than adding to it.
3. The m-fold composition is computed by a network with m hidden layers of three units each. How many hidden units would a network with a single hidden layer need to represent the same function? Compare the parameter counts of the two networks.

Adapted from Simon Prince, Understanding Deep Learning, Problem 4.8.

6. **One hidden layer suffices.** ▯ On the real line, a single hidden layer approximates a continuous function by a polyline, one hinge per joint. In the plane there are no joints to place, and the proof needs another idea: functions that vary along one direction only.

   1. Let ψ be continuous on $\mathbb{R}$ and let $\mathbf{w} \in \mathbb{R}^2$. Show that a single hidden layer approximates the *ridge function* $\psi(\mathbf{w}^\top \mathbf{x})$, which is constant along every line orthogonal to $\mathbf{w}$, uniformly on any compact subset of the plane.
   2. The Stone–Weierstrass theorem states the following. Let a set of continuous functions on a compact set be closed under sums, products, and scalar multiples, contain the constants, and contain, for any two distinct points, a function that takes different values at them. Then every continuous function on the compact set is a uniform limit of functions in the set. Show that the finite linear combinations of $\cos(\omega^\top \mathbf{x})$ and $\sin(\omega^\top \mathbf{x})$, with $\omega \in \mathbb{R}^2$, form such a set.
   3. Conclude that a single hidden layer approximates any continuous function on a compact subset of the plane.

   Adapted from Michael Nielsen, Neural Networks and Deep Learning, Chapter 4.

7. code **Weight scale and saturation.** ▯ Take the chain of Exercise 4 with $K = 20$, every weight $w_k = w$, and every bias $b_k = -w/2$, so that the input $h_0 = 1/2$ gives $z_k = 0$ at every layer.

   1. Express $\partial h_K / \partial h_0$ as a function of w for the input $h_0 = 1/2$. For which w does it not vanish as K grows?
   2. Implement the chain and run it from $h_0 = 0.6$ with $w = 4$ and with $w = 8$. Report h_{20} and $\partial h_{20}/\partial h_0$ in each case.
   3. Treat the map $h_{k-1} \mapsto h_k$ as a one-dimensional iteration with fixed point $1/2$, and use its slope there to explain the difference between the two runs. State what the result implies for countering vanishing gradients in a sigmoid network by scaling up its weights.

4.2 Implementation

In Section 4.1 we chose the weights of a hidden layer by hand, which is feasible for the four points of XOR and for little else. Let us now leave the choice to training, on real images.

On Fashion-MNIST, softmax regression reached about 83% validation accuracy after ten epochs, and most of its errors confused garments of similar outline, such as shirts, pullovers, and coats (Section 3.3). Telling such garments apart depends on how pixels combine, which is what a hidden layer can compute. How many of the errors does one hidden layer remove, which ones, and what does it cost? On paper the cost is considerable: with 256 hidden units the MLP has 26 times the parameters of softmax regression (Section 4.1) and performs 26 times the arithmetic on every image. Whether the gain in accuracy justifies this extra work is a question for experiment.

We implement the MLP twice and train it as we trained softmax regression. The first implementation writes the four parameter arrays, the activation function, and the forward pass with tensor operations, and it checks the untrained model before any training time is spent. The second assembles the same model from the framework's layers. The data, the cross-entropy loss, minibatch stochastic gradient descent (SGD), and the `Trainer` stay as they were, so any change in accuracy comes from the hidden layer. Finally we compare the MLP with softmax regression trained under the same settings, in accuracy, class by class, and in training time, and we look at what its hidden units compute. Automatic differentiation computes all gradients in this section; the next section shows how.

```
import time
```

4.2.1 Implementation from Scratch

4.2.1.1 The Parameters

Each 28×28 image becomes a vector of $d = 784$ pixel values, and there are $q = 10$ classes. We give the hidden layer $h = 256$ units (Figure 4.6). The width is a hyperparameter, like the learning rate: training does not choose it, and Exercise 4 compares other widths. In the notation of Section 4.1, each layer has a weight matrix with one row per input and one column per output, and a bias vector with one entry per output:

$$\mathbf{W}^{(1)} \in \mathbb{R}^{784\times256}, \quad \mathbf{b}^{(1)} \in \mathbb{R}^{256}, \quad \mathbf{W}^{(2)} \in \mathbb{R}^{256\times10}, \quad \mathbf{b}^{(2)} \in \mathbb{R}^{10}. \tag{4.12}$$

Together these hold the 203,530 numbers counted in Section 4.1. All but 2,570 of them belong to the hidden layer.

How should the four arrays start? As for softmax regression, we draw the weights from a normal distribution with mean 0 and standard deviation 0.01 and set the biases to 0. The weights must be random: hidden units that start with equal weights receive equal gradients and stay identical throughout training, as Section 4.4 shows. Small weights keep the logits close to 0 at first, so the untrained model gives every class a probability close to 1/10. The scale 0.01 itself is merely inherited from softmax regression, which keeps the comparison between the two models clean. It is not a good scale for a network with a hidden layer, and the first epochs of training will show it.

We register the four arrays as parameters of the module, as in Section 3.3. The default optimizer of `Classifier`, minibatch SGD at the learning rate `lr`, then finds them by itself, and the model needs no `configure_optimizers` of its own. The argument `std` sets the standard deviation of the initial weights.

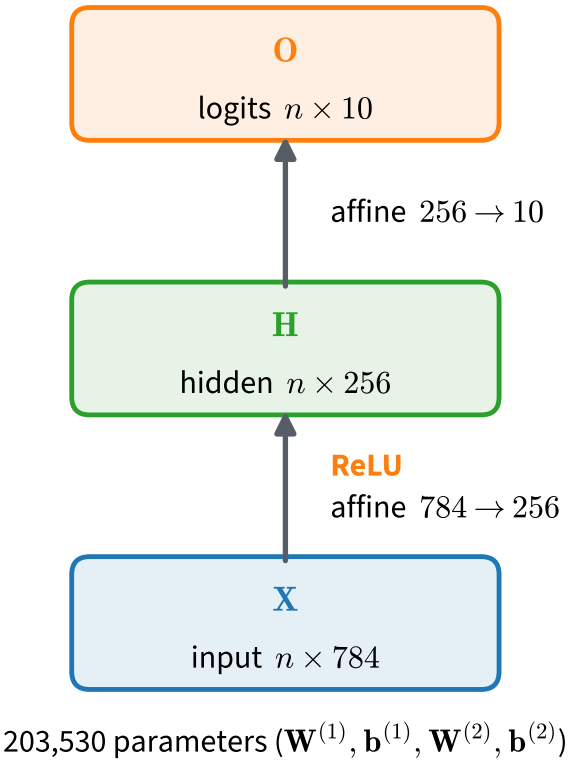


Figure 4.6: The MLP of this section. A minibatch of n images is flattened to an $n \times 784$ matrix $\mathbf{X}$, an affine map followed by the ReLU computes the $n \times 256$ hidden representation $\mathbf{H}$, and a second affine map computes the $n \times 10$ logits $\mathbf{O}$.

```
class MLPScratch(d2l.Classifier):
    """An MLP with one hidden layer, implemented from scratch."""
    def __init__(self, num_inputs, num_outputs, num_hiddens, lr, std=0.01):
        super().__init__()
        self.save_hyperparameters()
        self.W1 = nn.Parameter(d2l.normal(0, std, (num_inputs, num_hiddens)))
        self.b1 = nn.Parameter(d2l.zeros(num_hiddens))
        self.W2 = nn.Parameter(d2l.normal(0, std, (num_hiddens, num_outputs)))
        self.b2 = nn.Parameter(d2l.zeros(num_outputs))
```

4.2.1.2 The Forward Pass

Let us start with the activation function, which we write ourselves. The ReLU keeps the positive entries and zeros the others, so it multiplies each entry z by 1 if $z > 0$ and by 0 otherwise. (Written as a product with this mask, it also gets the derivative 0 at $z = 0$ from automatic differentiation, as the built-in ReLU does. A maximum with 0 may get 1 or $1/2$, depending on the framework, and Section 4.1.4 explained why we need not care.)

```
def relu(X):
    return X * (X > 0)
```

For a minibatch of n images, flattened into the rows of $\mathbf{X} \in \mathbb{R}^{n \times 784}$, the model computes

$$\mathbf{H} = \mathrm{ReLU}\left(\mathbf{X}\mathbf{W}^{(1)} + \mathbf{b}^{(1)}\right), \qquad \mathbf{O} = \mathbf{H}\mathbf{W}^{(2)} + \mathbf{b}^{(2)}. \tag{4.13}$$

Row i of $\mathbf{H} \in \mathbb{R}^{n \times 256}$ is the hidden representation of image i, and row i of $\mathbf{O} \in \mathbb{R}^{n \times 10}$ holds its logits; the biases are added to every row by broadcasting. The method `hidden` computes $\mathbf{H}$, and `forward` computes $\mathbf{O}$ from it. That is the whole model. Note that it returns logits, not probabilities: the loss that every `Classifier` inherits computes the cross-entropy from the logits (Section 3.4).

```
@d2l.add_to_class(MLPScratch)
def hidden(self, X):
    X = d2l.reshape(X, (-1, self.num_inputs))  # One flattened image per row
    return relu(d2l.matmul(X, self.W1) + self.b1)

@d2l.add_to_class(MLPScratch)
def forward(self, X):
    return d2l.matmul(self.hidden(X), self.W2) + self.b2  # The logits
```

4.2.1.3 Checks before Training

Training takes minutes, and a mistake in the model may only show as poor accuracy at the end. A forward pass through the untrained model on a few images tests three predictions in a fraction of a second. First, each image should have a hidden representation of 256 entries and one logit per class. Second, since the logits start close to 0, the loss should be close to $\log 10 \approx 2.30$, the cross-entropy of a prediction that gives every class the probability 1/10. Third, about half of the hidden activations should be positive. With zero biases, the

pre-activation of a hidden unit is a sum of pixel values times weights drawn symmetrically around 0, so it is positive or negative with equal probability, and the ReLU sets the negative ones to 0.

```
data = d2l.FashionMNIST(batch_size=256)
model = MLPScratch(num_inputs=784, num_outputs=10, num_hiddens=256, lr=0.1)
X, y = next(iter(data.train_dataloader()))
X, y = X[:8], y[:8]  # Eight images suffice for the checks
H, O = model.hidden(X), model(X)
active = d2l.reduce_mean(d2l.astype(H > 0, d2l.float32))
print('H:', H.shape, 'O:', O.shape)
print(f'loss {d2l.numpy(model.loss(O, y)):.3f}, '
      f'fraction of active units {d2l.numpy(active):.2f}')
```

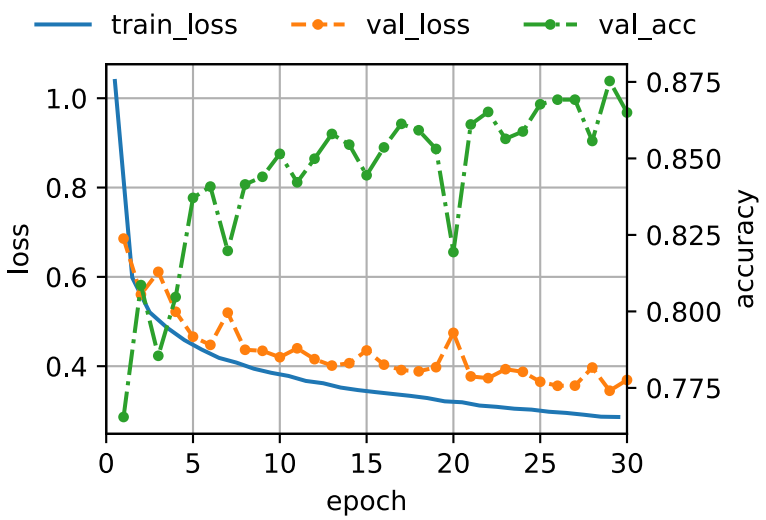
```
H: torch.Size([8, 256]) O: torch.Size([8, 10])
loss 2.303, fraction of active units 0.52
```

The shapes are 8×256 and 8×10, the loss is close to 2.30, and about half of the hidden units are active. A weight matrix with its axes swapped would stop the forward pass with an error, a wrong number of inputs would show up in the shapes, and a loss far from 2.30 would point to weights that are too large or to a mistake in the loss.

4.2.1.4 Training

We train with the minibatch size and the learning rate of softmax regression in Section 3.3, 256 images and 0.1, but for 30 epochs instead of 10. The longer run is a consequence of our initialization: with weights this small the hidden layer starts slowly, as the comparison at the end of this section shows. With `num_gpus=1` the `Trainer` trains on a GPU if the machine has one and on the CPU otherwise (Section 5.7).

```
trainer = d2l.Trainer(max_epochs=30, num_gpus=1)
trainer.fit(model, data)
```



The training loss falls over all 30 epochs. After the first few epochs the validation loss lies above it and falls more slowly, and the validation accuracy climbs for most of the run. The climb is far from smooth. From one epoch to the next the accuracy moves by about a point, and a single epoch can fall several points below its neighbors. Nothing is broken: minibatch SGD at a constant learning rate never settles, and we measure the validation accuracy wherever the last step of the epoch happened to leave the parameters.

4.2.2 Concise Implementation

Nobody who builds networks for a living registers four arrays by hand. A fully connected layer of the framework holds a weight matrix and a bias vector and computes the affine map of one layer, as in the concise softmax regression of Section 3.4. The concise MLP applies four operations in order: flattening, a fully connected layer with 256 outputs, the ReLU, and a fully connected layer with 10 outputs. The framework's sequential container holds the layers that perform them. We initialize the model like the scratch model. Left to its default, each framework would draw the weights from a distribution of its own, and the scratch and the concise model would start from different scales.

As for `SoftmaxRegression`, the lazy layers have no weights before the first forward pass, so the `Trainer` initializes them afterwards through the method `weight_init`.

```
class MLP(d2l.Classifier):
    """An MLP with one hidden layer, built from framework layers."""
    def __init__(self, num_outputs, num_hiddens, lr):
        super().__init__()
        self.save_hyperparameters()
        self.net = nn.Sequential(nn.Flatten(), nn.LazyLinear(num_hiddens),
                                 nn.ReLU(), nn.LazyLinear(num_outputs))

    def weight_init(self, module):
        # The Trainer calls this once the lazy layers know their shapes
        if isinstance(module, nn.Linear):
            nn.init.normal_(module.weight, std=0.01)
            nn.init.zeros_(module.bias)
```

The model needs no `forward` method of its own. It inherits the `forward` of `Module` (Section 2.4.2), which calls `self.net(X)`. Section 5.1.2 looks inside such a container.

The scratch model spelled out every array and every operation. Here the layers hold the parameters and the container fixes the order of the operations, and this is the form that scales: with a deeper network, or one whose layers we want to reorder, nothing has to be named and registered by hand.

4.2.3 Comparison with Softmax Regression

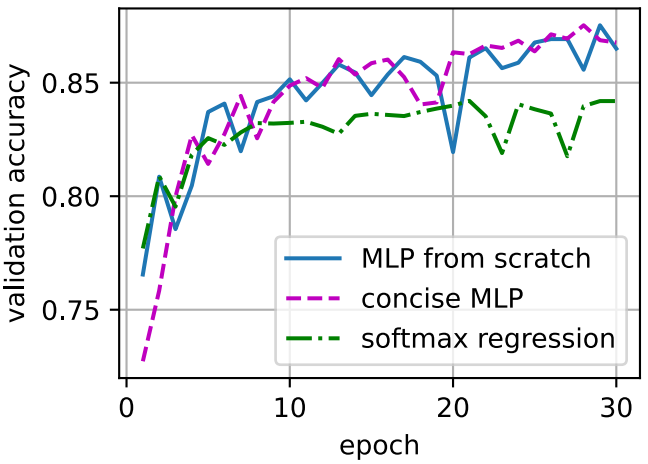
4.2.3.1 Accuracy

We now train the concise MLP and, for comparison, the concise softmax regression of Section 3.4, `d2l.SoftmaxRegression`, with the same trainer: 30 epochs, minibatches of 256, and learning rate 0.1. All three models start from the same initialization. We time each run and keep its validation accuracy after every epoch from `trainer.history`. The trainer's plot

is switched off, and we draw the validation curves of the three models, the scratch MLP included, in one plot instead.

```
curves, seconds = {'MLP from scratch': trainer.history['val_acc']}, {}
trainer.board.display = False
models = {'concise MLP': MLP(num_outputs=10, num_hiddens=256, lr=0.1),
          'softmax regression': d2l.SoftmaxRegression(num_outputs=10, lr=0.1)}
for name, net in models.items():
    start = time.perf_counter()
    trainer.fit(net, data)
    seconds[name] = (time.perf_counter() - start) / trainer.max_epochs
    curves[name] = trainer.history['val_acc']
```

```
d2l.plot(list(range(1, trainer.max_epochs + 1)), list(curves.values()),
         'epoch', 'validation accuracy', legend=list(curves))
```




```
print('validation accuracy averaged over the last 5 epochs')
for name, acc in curves.items():  # A single epoch can land a few points low
    print(f'  {name}: {sum(acc[-5:]) / 5:.3f}')
train_acc = trainer.evaluate(model, data, train=True)['acc']
print(f'training accuracy of the MLP from scratch: {train_acc:.3f}')
```

```
validation accuracy averaged over the last 5 epochs
  MLP from scratch: 0.867
  concise MLP: 0.870
  softmax regression: 0.836
training accuracy of the MLP from scratch: 0.893
```

The scratch and the concise MLP follow the same curve up to the variation from run to run, as they should: they are the same model, written in two ways. Both start more slowly than softmax regression and usually trail it by a few points after the first epoch. Our initialization is to blame. With weights of standard deviation 0.01, the hidden activations are small, and every gradient that reaches $\mathbf{W}^{(1)}$ is multiplied by entries of $\mathbf{W}^{(2)}$, which are just as small, so

at first the hidden layer learns slowly. After ten epochs, the training length of Section 3.3, the MLP is ahead by about a point on average, within its variation from epoch to epoch. Had we stopped there, we could hardly have told the two models apart. Since any single epoch can land a few points low, we compare the models by their validation accuracy averaged over the last five epochs: softmax regression levels off at about 84%, while the MLP climbs to about 87%. Three points sound modest. Read as errors, they are more: the hidden layer removes about a fifth of the errors of softmax regression. On its own training images the MLP is right about 89% of the time, only a few points more often than on the validation images.

4.2.3.2 Errors by Class

Which fifth? It is tempting to assume that it is the shirts taken for pullovers and coats, the errors we set out to remove. A total accuracy cannot tell us. We therefore count, for each class, the fraction of its validation images that a model classifies correctly. This is the *recall* of the class, the diagonal of the confusion matrix of Section 3.3. The table that follows has a useful feature: its two MLPs are the same model trained twice, so wherever their columns disagree, we see how much a single run varies.

```
def recall(net):
    """Fraction of the validation images of each class that net gets right."""
    correct, total = np.zeros(10), np.zeros(10)
    for X, y in map(trainer.prepare_batch, data.val_dataloader()):
        y_hat = d2l.numpy(d2l.argmax(net(X), axis=1))
        y = d2l.numpy(y).astype(int)
        total += np.bincount(y, minlength=10)
        correct += np.bincount(y, weights=y_hat == y, minlength=10)
    return correct / total

recalls = {'softmax': recall(models['softmax regression']),
           'MLP, scratch': recall(model),
           'MLP, concise': recall(models['concise MLP'])}
names = data.text_labels(range(10))
print(f'{"class":11s}' + ''.join(f'{name:>14s}' for name in recalls))
for j, name in enumerate(names):
    print(f'{name:11s}' + ''.join(f'{r[j]:14.2f}' for r in recalls.values()))
```

```
class             softmax  MLP, scratch  MLP, concise
T-shirt              0.81          0.80          0.73
trouser              0.95          0.96          0.96
pullover             0.75          0.91          0.86
dress                0.89          0.87          0.95
coat                 0.77          0.70          0.76
sandal               0.92          0.94          0.94
shirt                0.51          0.61          0.61
sneaker              0.93          0.95          0.95
bag                  0.95          0.97          0.96
ankle boot           0.95          0.95          0.95
```

The answer is less tidy than the question. Start with the six classes that softmax regression already recognized well: trousers, dresses, bags, and the three kinds of shoes. Their recall changes by a few points, more often up than down, and summed over the six, both MLPs

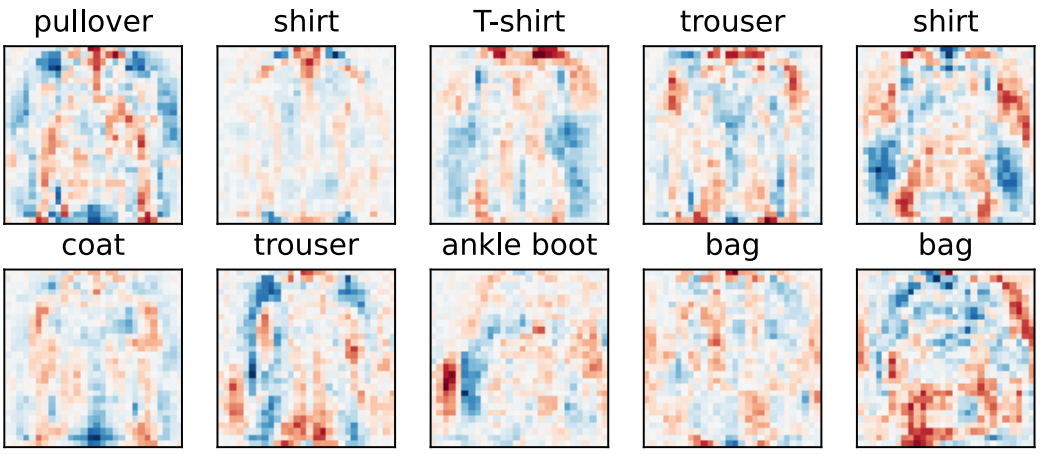
make fewer errors than softmax regression. Among the four tops (T-shirts, pullovers, coats, and shirts) the table is harder to read, because the two MLPs can disagree with each other by ten points or more on a single class, although they are the same model. What we see there is a boundary in motion. The last steps of SGD at a constant learning rate push the boundaries between similar garments back and forth, and what one top gains another loses. Only the sum is stable. Over the four tops together the MLP usually makes fewer errors than softmax regression, often by a few hundred, which is most of what it saves; after an unlucky last epoch it can even make more. So we cannot say that the shirts gain most, and neither does any other single class. The tops remain where most of the errors are.

4.2.3.3 The Learned Features

In Section 3.3 the weights themselves showed what softmax regression had learned: one template per class, each a blurred garment. For XOR we chose the hidden features by hand, and we said that for images training would have to choose them. What has it chosen? The weights tell again. Column j of $\mathbf{W}^{(1)}$ holds the weights of the 784 pixels into hidden unit j. Reshaped to 28×28, it is an image: a bright pixel where it is red raises the pre-activation of the unit, and a bright pixel where it is blue lowers it. Not every one of the 256 units makes a clear picture, so we draw the ten units of the scratch MLP with the largest incoming weights, each on its own color scale. Each is titled with the class whose logit it raises most, the largest entry of its row of $\mathbf{W}^{(2)}$.

```
W1, W2 = d2l.numpy(model.W1), d2l.numpy(model.W2)
```

```
units = np.argsort(np.linalg.norm(W1, axis=0))[::-1][:10]  # Largest weights
images = W1[:, units].T.reshape(10, 28, 28)
images /= np.abs(images).max(axis=(1, 2), keepdims=True)  # Own color scale
d2l.show_images(images, 2, 5, titles=[names[k] for k in W2[units].argmax(1)],
                cmap='RdBu_r', vmin=-1, vmax=1);
```



These are not ten more templates. Few of the units look like a whole garment. Most of them weigh a region: a neckline, the edge of a sleeve, the gap between the legs of a pair of trousers. Often red lies right beside blue, so that the unit responds to a contrast between neighboring

pixels and not to brightness alone. A template asks how much an image looks like a shirt. A hidden unit asks a smaller question, about one part of the image, and the output layer weighs 256 such answers. Note also the titles: most of the ten units raise the logit of one of the four tops, the shirt most often. It is at least suggestive that the units which training moved furthest serve the classes that softmax regression confused most, though ten units out of 256 do not prove it.

4.2.3.4 Parameters, Computation, and Time

The following table collects what the hidden layer adds. The multiply-adds are those of the forward pass for one image, $784{\cdot}10 = 7{,}840$ for softmax regression and $784{\cdot}256{+}256{\cdot}10 = 203{,}264$ for the MLP.

model	parameters	multiply-adds per image	validation accuracy, epochs 26–30
softmax regression	7,850	7,840	about 84%
MLP, 256 hidden units	203,530	203,264	about 87%

The hidden layer multiplies the parameters and the arithmetic by 26. Does it multiply the training time as well?

```
for name, s in seconds.items():
    print(f'{name}: {s:.1f} s per epoch')
```

```
concise MLP: 2.3 s per epoch
softmax regression: 2.4 s per epoch
```

Surprisingly, on a GPU it does not: the two times are about the same. For models this small, nearly all of an epoch goes to reading the images and to the fixed cost of every step, which do not grow with the width of the model; Exercise 4 measures how much.

The experiment leaves three questions open, and they are the three problems of the chapter introduction. Automatic differentiation computed the gradients of some two hundred thousand parameters without our looking inside; the next section derives them. Weights of standard deviation 0.01 slowed the start of training, and with more hidden layers they stop it altogether, as Section 4.4 will show. And a model with 3.4 parameters per training image could memorize its training set. Ours did nothing of the kind: it is hardly more accurate on the images it was fitted to than on those it never saw.

4.2.4 Summary

A multilayer perceptron with one hidden layer adds a weight matrix, a bias vector, and an activation function to softmax regression, and everything else in the training code stays as it was. From scratch, it consists of four registered parameter arrays, the ReLU written by hand, and a forward pass of two lines that returns logits. Before training, a forward pass through the untrained model checks the shapes, the initial loss, and the fraction of active hidden units against values we can predict. The concise version stacks the same operations in the framework's sequential container. On Fashion-MNIST, after 30 epochs, the hidden layer of 256 units removes about a fifth of the errors of softmax regression, with 26 times

the parameters and multiply-adds and, on a GPU, about the same training time per epoch. No single class gains reliably. The four tops still hold most of the errors, and the hidden units with the largest weights respond to parts of garments, not to whole ones.

4.2.5 Exercises

1. code **A hidden layer without activation.** Replace `relu` by the identity in `MLPScratch` and train the resulting model as in this section.
   1. Compare its validation accuracy after 30 epochs with that of the MLP and of softmax regression.
   2. After training, compute $\mathbf{W} = \mathbf{W}^{(1)}\mathbf{W}^{(2)}$ and $\mathbf{b} = \mathbf{W}^{(2)\top}\mathbf{b}^{(1)} + \mathbf{b}^{(2)}$, and verify on a minibatch that $\mathbf{XW} + \mathbf{b}$ reproduces the logits of the model.
   3. Exercise 1 showed that such a network computes exactly the affine maps whose weight matrix has rank at most the hidden width. Explain why this restriction costs nothing here, and state below which hidden width it would reduce what the model can represent on Fashion-MNIST.
2. code **Recall across runs.** The recall table of this section holds one run of softmax regression and two runs of the MLP, too few to say which differences between the models are real.
   1. Train each model five times and report, for every class, the mean and the range of its recall. For which classes is the difference between the two models larger than the spread within either of them?
   2. Compute the column-normalized confusion matrix $\mathbf{C}$ of Section 3.3 for one run of each model. Which confusions among the four tops does the hidden layer reduce, and which does it only move from one pair of classes to another?
   3. Keep a copy of the parameters of the MLP after each of its last five epochs, and classify each validation image by the average of the five predicted probabilities. Does the recall of the tops vary less between runs than that of a single epoch?
3. code **Single hidden unit.** Set `num_hiddens=1` in `MLPScratch`, so that its hidden layer has a single ReLU unit with weights $\mathbf{w}^{(1)} \in \mathbb{R}^{784\times 1}$ and bias $b^{(1)} \in \mathbb{R}$. Each input $\mathbf{x} \in \mathbb{R}^{1\times 784}$ is then reduced to the scalar $t = \mathrm{ReLU}(\mathbf{x}\mathbf{w}^{(1)} + b^{(1)}) \geq 0$, and the logits are $\mathbf{o} = t\,\mathbf{w}^{(2)} + \mathbf{b}^{(2)}$ with $\mathbf{w}^{(2)} \in \mathbb{R}^{1\times 10}$ and $\mathbf{b}^{(2)} \in \mathbb{R}^{10}$.
   1. Show that each class is predicted on at most one interval of t, and that the order of these intervals along the t axis is fixed by the entries of $\mathbf{w}^{(2)}$. Describe the regions of input space that the model assigns to the classes, and state what they imply for separating the ten Fashion-MNIST classes.
   2. Suppose that after some update $\mathbf{x}\mathbf{w}^{(1)} + b^{(1)} < 0$ for every training example. Which parameters receive nonzero gradients from then on, and what does the network predict? Why does the same event do little harm when it hits one of the 256 hidden units of this section?
   3. Train the model five times, and report for each run the validation accuracy and the fraction of validation examples on which $t > 0$, which `model.hidden` gives you. Compare any run in which this fraction is zero with your answer to (b).
4. code **Width.** Train `MLP` as in this section, for 30 epochs at learning rate 0.1, with hidden widths $h \in \{16, 64, 256, 1024, 4096\}$, and record the final validation accuracy and the time per epoch.
   1. Plot both against h on a logarithmic axis. Beyond which width does the accuracy stop improving by more than its variation from epoch to epoch?
   2. The multiply-adds grow in proportion to h. Time one pass over the training data

that only reads the minibatches, without any model, and use it to explain the part of the time curve that does not grow with h. From which width on does the arithmetic dominate on your hardware?

3. Repeat the training at widths 64 and 1024 with learning rates $\{0.03, 0.1, 0.3\}$. Does the best learning rate depend on the width?

5. code **Initialization scale and depth.** ☒ `MLPScratch` draws its weights with standard deviation 0.01, which its argument `std` sets.
   1. Train the one-hidden-layer model with standard deviations 0.001, 0.01, and 0.1, and plot the training and validation curves for each. How much do the curves differ, and does the difference persist as training continues? Relate your observation to the scale of the hidden activations and the logits on the first forward pass.
   2. Extend `MLPScratch` to two and to four hidden layers, each of width 256. Before running anything, predict which depth is more sensitive to the standard deviation, and why.
   3. Sweep the standard deviation over $\{0.001, 0.003, 0.01, 0.03, 0.1\}$ for both depths. For each depth, report the range for which the network reaches at least 70% validation accuracy within five epochs. Section 4.4 explains the pattern and trains a network with ten hidden layers.

Adapted from Stanford CS231n, Assignment 2, FullyConnectedNets Inline Question 1.

4.3 Backpropagation

In Section 4.2 we trained an MLP without writing a line of calculus: the framework supplied the gradients. Every step of minibatch stochastic gradient descent needed the derivative of the loss with respect to each of the network's 203,530 parameters, and automatic differentiation supplied all of them for a small multiple of the cost of the forward pass. How can one pass backward through a network compute so many derivatives at once?

Section 1.5 gave the answer in general terms: the framework records the forward pass as a computational graph and applies the chain rule backward through it, one local rule per operation, each reading values that the forward pass saved. There the backward pass through a network remained a black box, which this section opens. Little of the machinery is new. What is new is the setting: the parameters are matrices, a nonlinearity sits between the layers, and the gradient has to pass *through* one layer to reach the one below. An MLP needs only three local rules, for a matrix product, for adding a bias, and for an elementwise activation function, and applying them from the loss back to the first layer is backpropagation.

Three practical questions depend on the details. How much memory do the saved values take, and why does it grow with the minibatch? Why does a ReLU unit that is inactive for every example stop learning? And why do the gradients of a deep network vanish or explode, the subject of the next section? After a brief reminder on a graph of three numbers, we derive the three rules in matrix form, read the result as an error sent backward through the network, check it against automatic differentiation, and count what it costs in memory and time.

4.3.1 Computational Graphs

4.3.1.1 A Small Graph

The graph that we traced in Section 1.5 had vectors at its nodes. As a reminder of how a backward pass works, here are the two most common local rules in their barest form, on three numbers: $e = (a + b)\,c$ at $a = 2$, $b = 1$, and $c = -3$. The forward pass computes the intermediate value $u = a + b = 3$ and then $e = u\,c = -9$. The backward pass starts from $\partial e/\partial e = 1$ and multiplies by local derivatives on its way to the inputs. The multiplication has $\partial e/\partial u = c = -3$ and $\partial e/\partial c = u = 3$, the addition has $\partial u/\partial a = \partial u/\partial b = 1$, and the chain rule gives

$$\frac{\partial e}{\partial a} = \frac{\partial e}{\partial u}\,\frac{\partial u}{\partial a} = -3, \qquad \frac{\partial e}{\partial b} = \frac{\partial e}{\partial u}\,\frac{\partial u}{\partial b} = -3, \qquad \frac{\partial e}{\partial c} = u = 3.$$

An addition copies the incoming gradient to each of its inputs, and a multiplication scales it by the value of the *other* input. Figure 4.7 shows both passes.

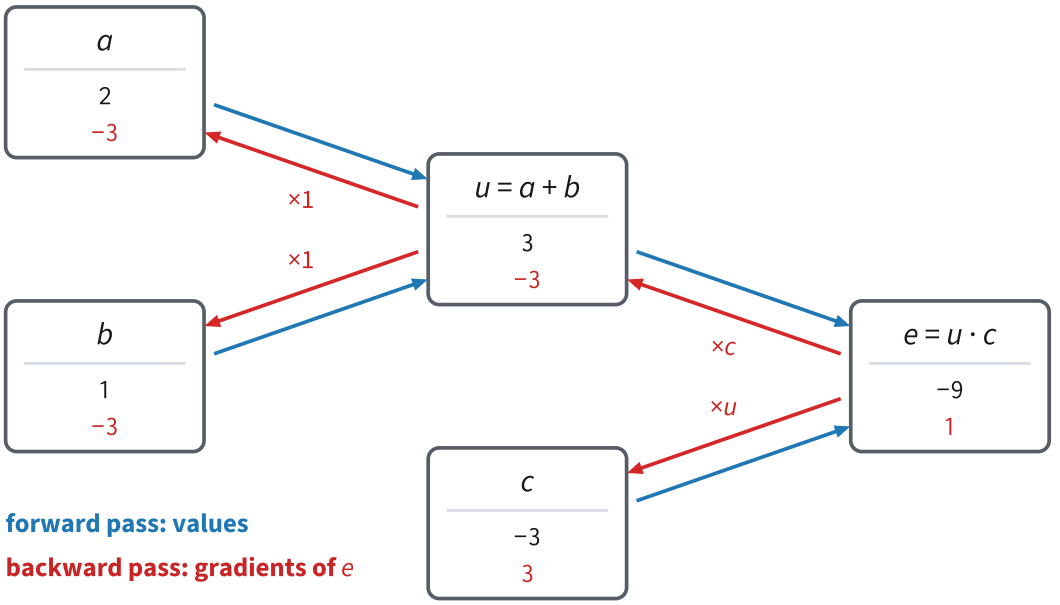


Figure 4.7: The graph of $e = (a + b)\,c$ at $a = 2$, $b = 1$, $c = -3$. Each node shows its value (black) and the derivative of e with respect to it (red). The forward pass (blue) runs from left to right; the backward pass (red) starts from 1 at e and multiplies by the local derivative written on each red arrow.

Both habits of a backward pass are already visible here. It reads values that the forward pass computed: the multiplication needed both of its inputs, c for the gradient of u and u for that of c, while the addition needed none. And it runs in the reverse order of the forward pass, here e first, then u and c, then a and b, since a gradient is ready only once all the nodes that use its value have passed theirs back.

4.3.1.2 The Graph of an MLP

Now to the network, the MLP of Section 4.1. A minibatch of n examples forms the rows of $\mathbf{X} \in \mathbb{R}^{n \times d}$. The hidden layer has weights $\mathbf{W}^{(1)} \in \mathbb{R}^{d \times h}$, biases $\mathbf{b}^{(1)} \in \mathbb{R}^{h}$, and activation function ϕ, and the output layer has weights $\mathbf{W}^{(2)} \in \mathbb{R}^{h \times q}$ and biases $\mathbf{b}^{(2)} \in \mathbb{R}^{q}$. The backward pass needs a name for the input of ϕ, the pre-activation $\mathbf{Z}$, so we write the forward pass in three steps:

$$\begin{aligned} \mathbf{Z} &= \mathbf{X}\mathbf{W}^{(1)} + \mathbf{b}^{(1)}, \\ \mathbf{H} &= \phi(\mathbf{Z}), \\ \mathbf{O} &= \mathbf{H}\mathbf{W}^{(2)} + \mathbf{b}^{(2)}. \end{aligned} \tag{4.14}$$

The first two lines are the hidden layer, and the third computes the logits. The loss is the cross-entropy of Section 3.1. Row i of $\hat{\mathbf{Y}} = \text{softmax}(\mathbf{O})$ holds the predicted class probabilities of example i, its loss $l_i = -\log \hat{y}_{i,y_i}$ is the negative log-probability of its true class y_i, and L is the average of l_i over the n examples. We also add weight decay (Section 2.7), which training usually includes and which makes each weight matrix feed two nodes, so that we see how gradients from several paths combine. Its penalty is

$$s = \frac{\lambda}{2} \left(\|\mathbf{W}^{(1)}\|_{\text{F}}^2 + \|\mathbf{W}^{(2)}\|_{\text{F}}^2 \right),$$

where the squared Frobenius norm $\|\mathbf{W}\|_{\text{F}}^2$ is the sum of the squares of all entries of $\mathbf{W}$. Training minimizes the *objective* $J = L + s$.

Figure 4.8 draws this computation as a graph. Its bottom row is the network itself: the input $\mathbf{X}$, the two values of the hidden layer, $\mathbf{Z}$ and $\mathbf{H}$, the logits $\mathbf{O}$, and the loss L. Each node is labeled with its shape for the MLP of Section 4.2, which has $d = 784$ inputs, $h = 256$ hidden units, and $q = 10$ classes. The weight matrices sit above the layers that use them, and each of them feeds two nodes: its layer and the penalty s.

4.3.2 Backpropagation through an MLP

4.3.2.1 Local Rules for Matrices

The backward pass of the MLP computes the gradient of the scalar J with respect to matrices. We write $\partial J/\partial \mathbf{W}$ for the matrix of partial derivatives $\partial J/\partial w_{rj}$, which has the same shape as $\mathbf{W}$. Each node of Figure 4.8 needs a rule that turns the gradient with respect to its output into gradients with respect to its inputs.

Let us start with the matrix product, the only one of the three rules that takes some care. In Section 1.5 a matrix–vector product passed the gradient back to its vector through $\mathbf{X}^\top$. Now both factors are matrices, and we need the gradient with respect to each. Consider $\mathbf{A} = \mathbf{X}\mathbf{W}$ with $\mathbf{X} \in \mathbb{R}^{n \times d}$ and $\mathbf{W} \in \mathbb{R}^{d \times h}$, inside a larger computation whose gradient $\mathbf{G} = \partial J/\partial \mathbf{A}$ is known. Entry $a_{ij} = \sum_r x_{ir} w_{rj}$ depends on w_{rj} with derivative x_{ir}, and w_{rj} enters every entry of column j of $\mathbf{A}$. The chain rule for many variables (Section 1.4) adds up the contributions of all these paths: $\partial J/\partial w_{rj} = \sum_i g_{ij} x_{ir}$, which is entry (r, j) of $\mathbf{X}^\top \mathbf{G}$. In the same way, x_{ir} enters every entry of row i, and $\partial J/\partial x_{ir} = \sum_j g_{ij} w_{rj}$. In matrix form,

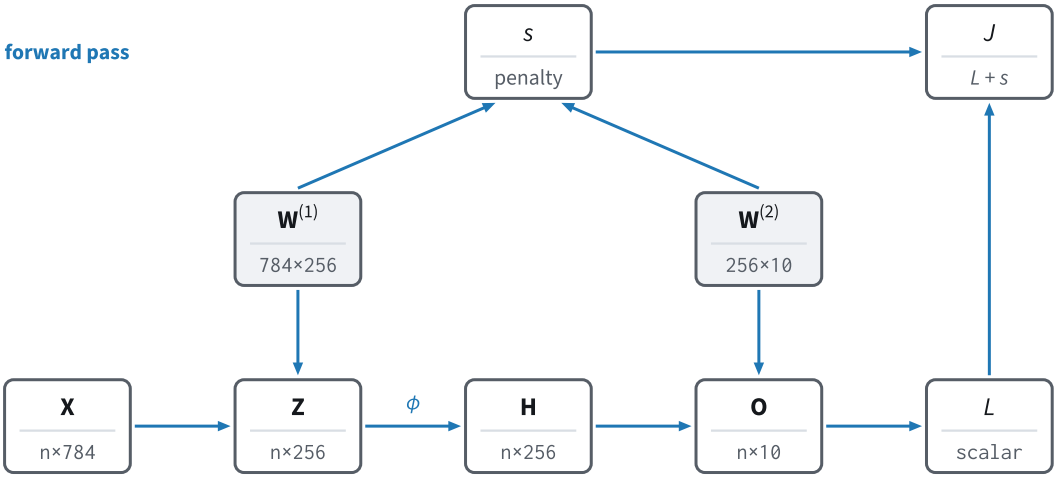


Figure 4.8: The forward pass of an MLP with one hidden layer and weight decay. Nodes are values, labeled with their shapes for $d = 784$, $h = 256$, and $q = 10$; parameters are shaded. The biases are omitted: they enter $\mathbf{Z}$ and $\mathbf{O}$ like the weights, but not the penalty s.

$$\frac{\partial J}{\partial \mathbf{W}} = \mathbf{X}^\top \mathbf{G}, \qquad \frac{\partial J}{\partial \mathbf{X}} = \mathbf{G}\,\mathbf{W}^\top. \tag{4.15}$$

Shapes give a quick check: $\mathbf{X}^\top\mathbf{G}$ is $d \times h$, like $\mathbf{W}$, and $\mathbf{G}\mathbf{W}^\top$ is $n \times d$, like $\mathbf{X}$, and when n, d, and h differ, no other product of the two factors has the right shape. Row i of $\mathbf{X}$ is $\mathbf{x}_i^\top$, the transposed input $\mathbf{x}_i \in \mathbb{R}^d$ of example i, and row i of $\mathbf{G}$ is $\mathbf{g}_i^\top$ with $\mathbf{g}_i \in \mathbb{R}^h$. The weight gradient $\mathbf{X}^\top\mathbf{G} = \sum_i \mathbf{x}_i\mathbf{g}_i^\top$ is therefore a sum of n outer products, one per example. The rule is the multiplication node of the small graph in matrix form. The gradient of each factor involves the *other* factor, so the forward pass must keep $\mathbf{X}$ for the gradient of $\mathbf{W}$. The second half of the rule, $\mathbf{G}\mathbf{W}^\top$, is what hands the gradient on to the layer below.

The other two operations are simpler. Adding the bias, $\mathbf{Z} = \mathbf{A} + \mathbf{b}$, adds b_j to every entry of column j. Like the addition of the small graph, it passes the incoming gradient $\mathbf{G} = \partial J/\partial \mathbf{Z}$ through unchanged, $\partial J/\partial \mathbf{A} = \mathbf{G}$, and the gradient of the bias is the sum of the rows of $\mathbf{G}$, one row per example. The activation $\mathbf{H} = \phi(\mathbf{Z})$ acts on each entry separately, so entry (i, j) of $\mathbf{H}$ depends on entry (i, j) of $\mathbf{Z}$ alone, and

$$\frac{\partial J}{\partial \mathbf{Z}} = \frac{\partial J}{\partial \mathbf{H}} \odot \phi'(\mathbf{Z}),$$

where $\odot$ multiplies matrices entry by entry. Each entry of the gradient is scaled by the slope of ϕ at the corresponding pre-activation. This rule needs $\mathbf{Z}$. For the ReLU, $\phi'(z)$ is 1 for $z > 0$ and 0 otherwise, so the rule passes the gradient of every active unit and blocks that of every inactive one. The ReLU can read the slope off its output, since $[\mathbf{H}]_{ij} > 0$ exactly when $[\mathbf{Z}]_{ij} > 0$. All three rules are special cases of the vector–Jacobian products of Section 1.5; the appendix derives them for general tensor operations in Section 34.4.3.2.

4.3.2.2 The Backward Pass

The three rules are all that the MLP needs. The backward pass starts at the objective with $\partial J/\partial J = 1$. Since $J = L + s$, both L and s receive the gradient 1. For the cross-entropy, Section 3.1 showed that the gradient of the loss of one example with respect to its logits is $\hat{\mathbf{y}} - \mathbf{y}$, the predicted probabilities minus the one-hot label. With the one-hot labels as the rows of $\mathbf{Y} \in \{0, 1\}^{n \times q}$ and the average over n examples, this gives the first line below. Each further line applies one of the local rules, following the arrows of Figure 4.8 backward from J, right to left along the bottom row:

$$\begin{aligned}
\frac{\partial J}{\partial \mathbf{O}} &= \frac{1}{n}(\hat{\mathbf{Y}} - \mathbf{Y}), \\
\frac{\partial J}{\partial \mathbf{W}^{(2)}} &= \mathbf{H}^\top \frac{\partial J}{\partial \mathbf{O}} + \lambda \mathbf{W}^{(2)}, \qquad \frac{\partial J}{\partial \mathbf{b}^{(2)}} = \left(\frac{\partial J}{\partial \mathbf{O}}\right)^\top \mathbf{1}, \\
\frac{\partial J}{\partial \mathbf{H}} &= \frac{\partial J}{\partial \mathbf{O}} \mathbf{W}^{(2)\top}, \\
\frac{\partial J}{\partial \mathbf{Z}} &= \frac{\partial J}{\partial \mathbf{H}} \odot \phi'(\mathbf{Z}), \\
\frac{\partial J}{\partial \mathbf{W}^{(1)}} &= \mathbf{X}^\top \frac{\partial J}{\partial \mathbf{Z}} + \lambda \mathbf{W}^{(1)}, \qquad \frac{\partial J}{\partial \mathbf{b}^{(1)}} = \left(\frac{\partial J}{\partial \mathbf{Z}}\right)^\top \mathbf{1}.
\end{aligned} \tag{4.16}$$

Here $\mathbf{1} \in \mathbb{R}^n$ is the vector of all ones, so $\mathbf{G}^\top \mathbf{1}$ sums the rows of $\mathbf{G}$ into a column vector, of the shape of the bias. Read from top to bottom, Equation 4.16 gives the order of the backward pass: the gradients reach the output layer first and the hidden layer last, and no line computes $\partial J/\partial \mathbf{X}$, because nothing below the input needs it. It also tells us what the forward pass must keep, namely the values that the lines read: $\hat{\mathbf{Y}}$, $\mathbf{H}$, $\mathbf{W}^{(2)}$, $\mathbf{Z}$, and $\mathbf{X}$.

The display also has a meaning, and its third line is where the method gets its name. The first line is the error at the outputs: the predicted probabilities minus the labels. The third line sends this error back through the same weights that carried the hidden activations forward, so that hidden unit j receives the errors of the q outputs, each multiplied by the weight from unit j to that output. A hidden unit has no label of its own. What it learns from is this message: how wrong the outputs that it feeds are, and in which direction. The fourth line lets the message through only where the slope of the unit's activation is not zero.

The weights receive two terms. $\mathbf{W}^{(2)}$ reaches J along two paths in Figure 4.8, through $\mathbf{O}$ and L and through the penalty s, and the chain rule adds the contributions of all paths, as it did for the product $z = ux$ in Section 1.5: $\mathbf{H}^\top \partial J/\partial \mathbf{O}$ from the first and $\lambda \mathbf{W}^{(2)}$ from the second. A hand-written backward pass that assigns the second contribution instead of adding it, `=` where `+=` belongs, computes a wrong gradient without any error message.

Nothing here is special to a single hidden layer. Let layer k of K compute $\mathbf{Z}^{(k)} = \mathbf{H}^{(k-1)}\mathbf{W}^{(k)} + \mathbf{b}^{(k)}$, with $\mathbf{H}^{(0)} = \mathbf{X}$, $\mathbf{H}^{(k)} = \phi(\mathbf{Z}^{(k)})$ in the hidden layers, and the logits $\mathbf{O} = \mathbf{Z}^{(K)}$. The backward pass starts as before, from $\partial J/\partial \mathbf{Z}^{(K)} = \frac{1}{n}(\hat{\mathbf{Y}} - \mathbf{Y})$, and repeats one step from the last hidden layer down to the first:

$$\frac{\partial J}{\partial \mathbf{Z}^{(k)}} = \left(\frac{\partial J}{\partial \mathbf{Z}^{(k+1)}} \mathbf{W}^{(k+1)\top}\right) \odot \phi'(\mathbf{Z}^{(k)}), \qquad \frac{\partial J}{\partial \mathbf{W}^{(k)}} = \mathbf{H}^{(k-1)\top} \frac{\partial J}{\partial \mathbf{Z}^{(k)}} + \lambda \mathbf{W}^{(k)}.$$

The first equation merges lines 3 and 4 of Equation 4.16; the second is the rule for the weights, which holds for the last layer as well. A layer need not know how many others there are. It receives the error of the layer above, takes its own gradients from it, and passes it on, and on the way down the error is multiplied by one transposed weight matrix and one matrix of slopes per layer.

4.3.2.3 A Worked Example

We now run Equation 4.16 by hand on a network small enough to check every number: $d = h = 2$ inputs and hidden units, $q = 1$ output, a single example, ReLU activation, zero biases, and no weight decay. The output is a single number o, and the loss is the squared error $L = \frac{1}{2}(o - y)^2$. We take

$$\mathbf{X} = [1 \quad 2], \quad \mathbf{W}^{(1)} = \begin{bmatrix} 1 & 0 \\ -1 & 1 \end{bmatrix}, \quad \mathbf{W}^{(2)} = \begin{bmatrix} 2 \\ -1 \end{bmatrix}, \quad y = 0. \tag{4.17}$$

We treat the single example as a minibatch with $n = 1$, so $\mathbf{Z}$, $\mathbf{H}$, and their gradients are 1×2 matrices, rows of two numbers like $\mathbf{X}$. The forward pass gives $\mathbf{Z} = \mathbf{X}\mathbf{W}^{(1)} = [-1 \quad 2]$ and $\mathbf{H} = [0 \quad 2]$: the first hidden unit is inactive. Then $o = \mathbf{H}\mathbf{W}^{(2)} = -2$ and $L = \frac{1}{2}(-2)^2 = 2$. For the squared loss, only the first line of Equation 4.16 changes: $\partial L/\partial o = o - y = -2$. The other lines give

$$\frac{\partial L}{\partial \mathbf{W}^{(2)}} = \mathbf{H}^\top \frac{\partial L}{\partial o} = \begin{bmatrix} 0 \\ 2 \end{bmatrix} (-2) = \begin{bmatrix} 0 \\ -4 \end{bmatrix}, \qquad \frac{\partial L}{\partial \mathbf{H}} = \frac{\partial L}{\partial o} \mathbf{W}^{(2)\top} = [-4 \quad 2],$$
$$\frac{\partial L}{\partial \mathbf{Z}} = [-4 \quad 2] \odot [0 \quad 1] = [0 \quad 2], \qquad \frac{\partial L}{\partial \mathbf{W}^{(1)}} = \mathbf{X}^\top \frac{\partial L}{\partial \mathbf{Z}} = \begin{bmatrix} 1 \\ 2 \end{bmatrix} [0 \quad 2] = \begin{bmatrix} 0 & 2 \\ 0 & 4 \end{bmatrix}.$$

Column j of $\mathbf{W}^{(1)}$ holds the weights into hidden unit j, and the first column of $\partial L/\partial \mathbf{W}^{(1)}$ is zero: the ReLU passed no gradient back to the inactive unit, so this example does not change the weights into it. A unit that is inactive for every training example receives no gradient from any of them: this is the dead unit of Section 4.1, seen in its gradient. The bias gradients follow from the rows that we already have. For a single example, the sum over rows is the row itself, so $\partial L/\partial \mathbf{b}^{(1)} = (0, 2)$ holds the entries of $\partial L/\partial \mathbf{Z}$, and $\partial L/\partial \mathbf{b}^{(2)} = \partial L/\partial o = -2$. Figure 4.9 shows all the values and gradients in the graph.

4.3.2.4 Checking the Backward Pass on a Minibatch

The worked example was chosen to be easy: a single example, the squared loss, no weight decay. To test every line of Equation 4.16, we implement it for the full MLP and compare the result with automatic differentiation. The function `mlp_backward` runs the forward pass Equation 4.14, keeping the values that the backward pass reads, and then the five lines of Equation 4.16, with ReLU as ϕ.

```
def mlp_backward(X, Y, W1, b1, W2, b2, lambd):
    """Gradients of J = cross-entropy + weight decay, by backprop."""
    Z = X @ W1 + b1  # forward pass, keeping X, Z, H, and Y_hat
    H = torch.relu(Z)
    Y_hat = torch.softmax(H @ W2 + b2, dim=1)
```

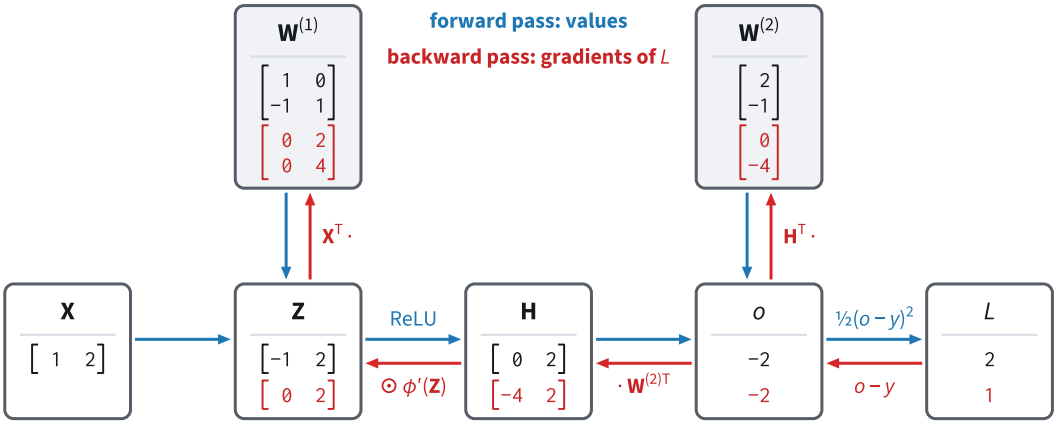


Figure 4.9: The worked example as a graph. Each node shows its value (black) and the gradient of L with respect to it (red); the biases are zero and omitted. The forward pass (blue) runs left to right and down into each layer; the backward pass (red) applies the rule written on each red arrow. The inactive first hidden unit, $z_1 = -1$, receives no gradient, which zeros the first column of $\partial L/\partial \mathbf{W}^{(1)}$.

```
    G_O = (Y_hat - Y) / X.shape[0]  # dJ/dO
    dW2, db2 = H.T @ G_O + lambd * W2, G_O.sum(dim=0)
    G_Z = (G_O @ W2.T) * (Z > 0)  # dJ/dH, then dJ/dZ
    dW1, db1 = X.T @ G_Z + lambd * W1, G_Z.sum(dim=0)
    return dW1, db1, dW2, db2
```

We draw a minibatch of 256 random inputs and labels, initialize the parameters as in Section 4.2, with weights from $\mathcal{N}(0, 0.01^2)$ and zero biases, and set $\lambda = 10^{-3}$. Automatic differentiation differentiates the framework's own cross-entropy loss. As is usual for gradient checks, we compute in 64-bit floating point, whose rounding error, about 10^{-16} relative to each result, is far smaller than the error that a wrong line would cause. For each parameter we print the largest difference between the two gradients, relative to the largest entry of the gradient.

```
n, d, h, q, lambd = 256, 784, 256, 10, 1e-3
f64 = torch.float64  # 64-bit floating point throughout the check
X, y = torch.randn(n, d, dtype=f64), torch.randint(0, q, (n,))
params = [torch.randn(d, h, dtype=f64) * 0.01, torch.zeros(h, dtype=f64),
          torch.randn(h, q, dtype=f64) * 0.01, torch.zeros(q, dtype=f64)]
for p in params:
    p.requires_grad_(True)
W1, b1, W2, b2 = params
J = (F.cross_entropy(torch.relu(X @ W1 + b1) @ W2 + b2, y)
     + lambd / 2 * ((W1 ** 2).sum() + (W2 ** 2).sum()))
J.backward()  # automatic differentiation
with torch.no_grad():
    grads = mlp_backward(X, F.one_hot(y, q).to(f64), *params, lambd)
for name, p, g in zip(['W1', 'b1', 'W2', 'b2'], params, grads):
```

```
    rel = (p.grad - g).abs().max() / p.grad.abs().max()
    print(f'{name}: relative difference {float(rel):.1e}')
```

```
W1: relative difference 4.4e-16
b1: relative difference 2.9e-16
W2: relative difference 3.8e-16
b2: relative difference 2.1e-16
```

All relative differences are below 10^{-14}, within a few rounding errors of 64-bit floating point: for this network, Equation 4.16 is the computation that automatic differentiation performs.

4.3.3 Memory and Computation

4.3.3.1 Stored Values

The backward pass reads values that the forward pass computed, and Section 1.5 named this as the reason why training needs more memory than prediction. For the MLP of Section 4.2 we can say how much more. The lines of Equation 4.16 read $\mathbf{X}$, $\mathbf{H}$, and $\hat{\mathbf{Y}}$, and with the ReLU, $\mathbf{H}$ also determines $\phi'(\mathbf{Z})$, so $\mathbf{Z}$ need not be kept. A minibatch of $n = 256$ examples has

$$256 \times 784 + 256 \times 256 + 256 \times 10 = 200{,}704 + 65{,}536 + 2{,}560 = 268{,}800 \qquad (4.18)$$

such numbers, which take 1.1 MB in 32-bit floating point at 4 bytes each. The network has only 203,530 parameters, 0.8 MB. Even for this small network, then, the values saved for one minibatch outweigh the model. Unlike the parameters, they grow in proportion to the minibatch size n, and every additional hidden layer of width 256 adds another 256×256 matrix per minibatch. Prediction stores nothing for later: it can discard each layer's input as soon as the layer's output is computed. The largest minibatch that fits on a GPU is therefore often set by the stored values and not by the parameters.

4.3.3.2 Computation

What about time? Almost all of the work lies in the matrix products. For each example, the forward pass of the MLP computes $784 \times 256 + 256 \times 10 = 203{,}264$ multiply-adds. Each forward matrix product turns into two backward products of the same size, Equation 4.15, one for the weights and one for the input, except that the first layer needs no gradient for its input $\mathbf{X}$. The backward pass therefore computes $784 \times 256 + 2 \times 256 \times 10 = 205{,}824$ multiply-adds per example. A training step costs about two forward passes for this network and about three for a network whose first layer is small, since then nearly every product appears twice in the backward pass.

This settles the question we started with. Finite differences would need one forward pass per parameter, some two hundred thousand per training step. The backward pass computes $\partial J/\partial \mathbf{O}$ and $\partial J/\partial \mathbf{Z}$ once, and the derivative of the loss with respect to every parameter is read off these two matrices by a single product or sum: the work that the derivatives have in common is done once. This is the advantage of reverse mode (Section 1.5), and here we can point to the shared quantities: they are the errors of the layers. In a deep network the error passes through many layers, and the next section asks what a long product of transposed weight matrices and slopes does to its size.

4.3.4 Summary

Backpropagation is the reverse mode of Section 1.5 applied to a network. Three local rules suffice for an MLP: an addition passes the incoming gradient through, a matrix product $\mathbf{XW}$ passes back $\mathbf{X}^\top\mathbf{G}$ to the weights and $\mathbf{GW}^\top$ to its input, and an elementwise activation multiplies by its slope $\phi'(\mathbf{Z})$, which for the ReLU is zero at every inactive unit. A value that feeds several nodes adds up the gradients that they pass back. Each layer receives the error of the layer above, sent back through the weights between them, and every parameter gradient is read off these shared per-layer errors, which is why one pass suffices. The rules read values from the forward pass, which training must store: for the MLP of Section 4.2, 1.1 MB per minibatch of 256 examples, more than its parameters, and growing with the minibatch size and the depth.

4.3.5 Exercises

1. **Jacobian of a layer.** The gradient of the scalar J with respect to $\mathbf{W}^{(2)} \in \mathbb{R}^{h\times q}$ has the shape of $\mathbf{W}^{(2)}$. The derivative of the matrix $\mathbf{O} = \mathbf{HW}^{(2)} \in \mathbb{R}^{n\times q}$ with respect to $\mathbf{W}^{(2)}$ is a larger object, with one entry $\partial o_{ij}/\partial w_{rt}$ for every combination of the four indices.
   1. How many entries does $\partial\mathbf{O}/\partial\mathbf{W}^{(2)}$ have, and how many of them can be nonzero?
   2. Evaluate the chain rule $\partial J/\partial w_{rt} = \sum_{i,j}(\partial J/\partial o_{ij})(\partial o_{ij}/\partial w_{rt})$ to recover $\mathbf{H}^\top\partial J/\partial\mathbf{O}$.
   3. How much memory would $\partial\mathbf{O}/\partial\mathbf{W}^{(2)}$ take in 32-bit floating point for the MLP of Section 4.2 with $n = 256$? Explain why backpropagation never forms it.
2. **Modified activation.** The first hidden unit of the network in Equation 4.17 receives the pre-activation $z_1 = -1$, so the ReLU derivative $\phi'(z_1) = 0$ blocks the gradient of every weight into this unit. Replace the activation of this unit alone by the leaky variant $\phi_1(z) = \max(0, z) + 0.1\min(0, z)$; the second unit keeps its ReLU.
   1. Write the backward rule for $\partial J/\partial\mathbf{Z}$ when hidden unit j has its own activation ϕ_j.
   2. Redo the forward and backward passes, and give $\partial L/\partial\mathbf{W}^{(2)}$, $\partial L/\partial\mathbf{Z}$, and $\partial L/\partial\mathbf{W}^{(1)}$. Which of these gradients differ from the all-ReLU case even though the rules that produce them are unchanged, and why?

   Adapted from Michael Nielsen, Neural Networks and Deep Learning, Chapter 2, and Simon Prince, Understanding Deep Learning, Problem 7.10.
3. **Memory of a deeper network.** Consider an MLP with d inputs, q outputs, and K layers, of which the $K-1$ hidden ones are ReLU layers of width h, trained on minibatches of n examples.
   1. Using the step for K layers in the text, list the values that the backward pass reads, and count them. Count the parameters.
   2. Prediction can discard each value as soon as the next layer has used it. Count the largest number of values that prediction holds at any moment.
   3. For $d = 784$, $h = 256$, and $q = 10$, find the smallest minibatch size at which the stored values outnumber the parameters, for $K = 2$ and for $K = 11$.
4. code **Gradient checking.** Before automatic differentiation, a hand-written backward pass was checked against finite differences.
   1. Pick one entry w of $\mathbf{W}^{(1)}$ and compare its gradient from `mlp_backward` with the central difference $(J(w+\epsilon) - J(w-\epsilon))/(2\epsilon)$ for $\epsilon = 10^{-1}, 10^{-2}, \ldots, 10^{-7}$, once in 32-bit and once in 64-bit floating point. Plot the relative error against ϵ on logarithmic axes.
   2. Explain the shape of the curves. Which error decreases with ϵ, which increases,

and where do they balance in each precision?
3. The check can fail at a correct gradient when a pre-activation lies within ϵ of zero. Explain why, and how to avoid it.

5. **Forward mode.** Backpropagation seeds $\partial J/\partial J = 1$ at the output and sweeps the graph backward. Forward-mode differentiation instead seeds $\partial w/\partial w = 1$ at one chosen scalar parameter w and sweeps forward: each node computes the derivative of its value with respect to w from the derivatives of its inputs.
   1. Use forward mode to recompute $\partial L/\partial \mathbf{W}^{(1)}$ for the network in Equation 4.17, one sweep per entry of $\mathbf{W}^{(1)}$.
   2. How many sweeps does each mode need to obtain all six weight gradients of this network? Which products does forward mode compute again in every sweep that reverse mode computes only once?

   Adapted from Simon Prince, Understanding Deep Learning, Problems 7.12 and 7.13.
6. **Hessian of a ReLU network.** ReLU has $\phi''(z) = 0$ for $z \neq 0$. For the network in Equation 4.17, derive the 4×4 Hessian of L with respect to the entries of $\mathbf{W}^{(1)}$, show that it has rank at most one, and evaluate it at the given values.
7. [extended] **Autograd engine.** Build a miniature automatic differentiation engine and use it to recompute the gradients of the network in Equation 4.17.
   1. Write a scalar `Value` class that records, for each result, its inputs and the operation that produced it (the computational graph), supporting `+`, `*`, and `relu`. (*Hint:* implement `__add__` and `__mul__` so that each returns a new `Value` holding references to its parents and a small function that propagates the gradient one step.)
   2. Implement `backward()`: sort the graph topologically, seed the output's gradient with 1, and visit the nodes in reverse order, letting each node pass its gradient to its parents. Make sure that gradients *accumulate* (`+=`, not `=`) when a value is used more than once.
   3. Recompute the graph of Figure 4.7 and the worked example Equation 4.17 with your engine (unroll the matrix products into scalars), and compare the gradients with those derived in this section.
   4. Consider the chain in which a scalar input x produces $v_i = f_i(x)$ for $i = 1, 2, 3$, the intermediate value $m = v_1 + v_2 + v_3$, the values $p_j = g_j(m)$ for $j = 1, 2, 3$, and the loss $L = p_1 + p_2 + p_3$. Write $\alpha_i = f_i'(x)$ and $\beta_j = g_j'(m)$. The chain rule as a sum over paths enumerates $3 \times 3 = 9$ paths from x to L, each contributing the product $\alpha_i \beta_j$. Show that your engine instead computes $(\alpha_1 + \alpha_2 + \alpha_3)(\beta_1 + \beta_2 + \beta_3)$ by accumulating the gradient at m once, and explain why this sharing keeps backpropagation affordable in deep networks.

   Adapted from Andrej Karpathy, micrograd, `micrograd/engine.py`.

4.4 Numerical Stability and Initialization

Depth is worth having (Section 4.1). Yet the MLP we trained in Section 4.2 has a single hidden layer, and we drew its initial weights from a normal distribution with standard deviation 0.01 without asking whether that choice matters. Would a deeper network fare as well?

Consider a network with ten hidden layers of width 256, initialized the same way. It does not learn at all: after ten epochs its loss is where it started. In fact it is in trouble before the first step of training. The backward pass of Section 4.3 multiplies the gradient by one weight matrix per layer on its way down to the first layer, and with weights this small each multiplication shrinks the mean square of the gradient about eightyfold. After nine such

steps the mean square at the first layer is 10^{17} times smaller than at the top, so a typical entry of the gradient, its square root, is smaller by a factor of more than 10^8. The signal fares no better on its way up: the same factor shrinks the activations from layer to layer, and what arrives at the last hidden layer is practically zero for every input. With weights of standard deviation 1, both grow at a similar rate instead.

The right scale depends on the width of each layer and on the activation function. We measure how the gradient shrinks or grows from layer to layer, derive the scales that keep it constant, known as Xavier and He initialization, and check them with the same measurement. All of these measurements are taken on randomly initialized networks. At the end we train the deep network with each initialization and see whether they predicted its fate.

A second requirement, stated without proof in Section 4.2, cannot be met by any choice of scale: the initial weights of different hidden units must differ, or the units remain identical throughout training.

4.4.1 Vanishing and Exploding Gradients

4.4.1.1 A Product over Layers

Why should the scale of the weights matter so much? The backward pass of Section 4.3 holds the answer: on its way down, the error is multiplied by one transposed weight matrix and one matrix of slopes per layer. Let us write this product out, in the notation of that section. The MLP has K layers, $K-1$ of them hidden and of width h. Starting from $\mathbf{H}^{(0)} = \mathbf{X}$, with one example per row, hidden layer k computes the pre-activation $\mathbf{Z}^{(k)} = \mathbf{H}^{(k-1)}\mathbf{W}^{(k)} + \mathbf{b}^{(k)}$ and the activation $\mathbf{H}^{(k)} = \phi(\mathbf{Z}^{(k)})$. There is no weight decay in this section, so the objective J of Section 4.3 is the loss L itself. We follow the gradient of L with respect to the activations. The rule for an activation function and the rule for a matrix product, applied in turn, take it one layer down:

$$\frac{\partial L}{\partial \mathbf{H}^{(k-1)}} = \left(\frac{\partial L}{\partial \mathbf{H}^{(k)}} \odot \phi'\big(\mathbf{Z}^{(k)}\big) \right) \mathbf{W}^{(k)\top}. \tag{4.19}$$

This is the step for K layers of Section 4.3, written for the activations instead of the pre-activations: multiply entry by entry by the slopes, then by the transposed weights. Repeating it from the last hidden layer, $K-1$, down to layer k gives

$$\frac{\partial L}{\partial \mathbf{H}^{(k)}} = \frac{\partial L}{\partial \mathbf{H}^{(K-1)}} \underbrace{\odot\, \phi'\big(\mathbf{Z}^{(K-1)}\big)\, \mathbf{W}^{(K-1)\top}}_{\text{layer } K-1} \cdots \underbrace{\odot\, \phi'\big(\mathbf{Z}^{(k+1)}\big)\, \mathbf{W}^{(k+1)\top}}_{\text{layer } k+1}, \tag{4.20}$$

where the operations are applied from left to right: one pair of them for each of the $K-1-k$ layers above layer k. The gradient with respect to the weights of layer k, $\mathbf{H}^{(k-1)\top}\big(\partial L/\partial \mathbf{H}^{(k)} \odot \phi'(\mathbf{Z}^{(k)})\big)$, contains this product as a factor.

A product of many factors is very large or very small unless the factors are close to 1. If each factor scales what it multiplies by about the same amount ρ, a product of t of them scales it by about ρ^t. We saw the same effect for repeated matrix products in Section 1.3. With $\rho = 0.9$ the gradient shrinks 200-fold over fifty layers, and with $\rho = 1.1$ it grows 117-fold.

So which factor does one layer contribute? This is the key computation of the section, and it is short. Every row of Equation 4.19 belongs to one example, and the rows do not mix, so we

look at a single one. Write $\mathbf{g}$ for this row of the gradient below layer k, $\mathbf{g}'$ for the same row above it, and z_j for the pre-activation of unit j. Entry i of Equation 4.19 is a sum over the h units of the layer, $g_i = \sum_j w_{ij}\,\phi'(z_j)\,g'_j$, and its square is a double sum over j and j' with terms $w_{ij}w_{ij'}\,\phi'(z_j)\phi'(z_{j'})\,g'_j g'_{j'}$. Suppose that the weights are drawn independently with mean zero and variance τ^2, and that they are independent of the slopes and of the gradient above. Then every term with $j \neq j'$ has mean zero, since w_{ij} and $w_{ij'}$ do. Only the h terms with $j = j'$ survive, each with mean $\tau^2\,\mathbb{E}[\phi'(z_j)^2\,(g'_j)^2]$. Treating the slope and the gradient above as independent as well, we find $\mathbb{E}[g_i^2] = h\,\tau^2\,\mathbb{E}[\phi'(z)^2]\,\mathbb{E}[(g'_j)^2]$. Per layer, the mean square of the gradient is therefore multiplied by about

$$h\,\tau^2\,\mathbb{E}\big[\phi'(z)^2\big]. \tag{4.21}$$

The word "about" is there for a reason. Independence of the weights and the slopes is a rather bold assumption, since the slope $\phi'(z_j)$ was computed in the forward pass from the very weights that now multiply it. It is plausible in a wide layer, where w_{ij} is only one of the h weights that enter z_j. We do not prove it; the measurement below is its test.

Two things set the factor: the scale of the weights, through $h\tau^2$, and the slope of the activation function, through $\mathbb{E}[\phi'(z)^2]$. When the pre-activations are symmetric about zero, the ReLU has slope 1 for half of them and 0 for the other half, so $\mathbb{E}[\phi'(z)^2] = 1/2$. For a ReLU layer of width $h = 256$ with the weights of Section 4.2, $\tau = 0.01$, the factor is $256 \times 10^{-4}/2 = 0.0128$; with $\tau = 1$ it is 128.

We measure these factors in a network of ten hidden layers of width 256, without biases, at initialization. The function `moments` runs the forward pass on a minibatch of 256 random inputs, keeping the pre-activations, and then the backward pass. We start the backward pass at the last hidden layer from a random gradient of mean square 1, so that the plot shows only what the layers below do to it. Each step of its backward loop is one line of code, and that line is Equation 4.19. The function returns the mean squares of the activations and of the gradients at every layer. Each activation function is passed together with its derivative.

```
def moments(phi, dphi, scale, depth=10, width=256, n=256):
    """Mean squares of the activations and of their gradients, per layer."""
    Ws = [scale * torch.randn(width, width) for _ in range(depth)]
    H, Zs, acts = torch.randn(n, width), [], []
    for W in Ws:  # forward pass, keeping the pre-activations
        Zs.append(H @ W)
        H = phi(Zs[-1])
        acts.append(float((H ** 2).mean()))
    G, grads = torch.randn(n, width), []  # dL/dH at the top layer
    for W, Z in zip(reversed(Ws), reversed(Zs)):  # backward pass
        grads.insert(0, float((G ** 2).mean()))
        G = (G * dphi(Z)) @ W.T  # dL/dH one layer down
    return acts, grads

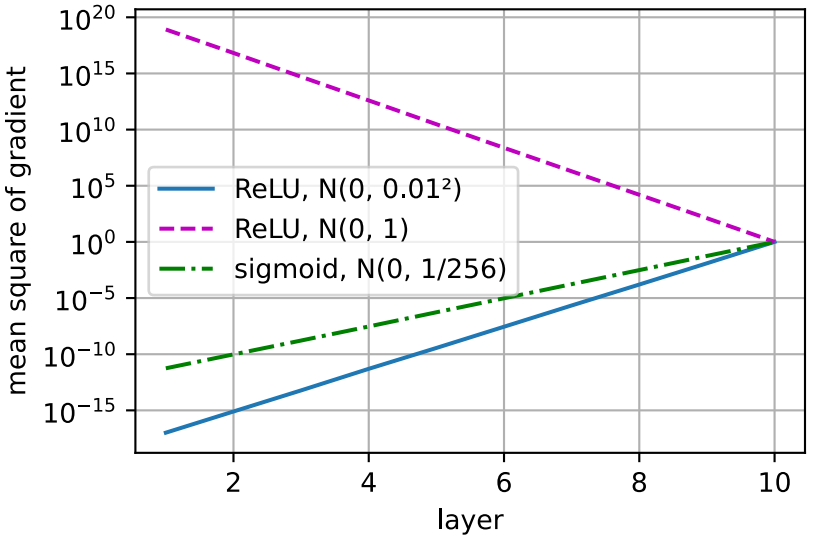
relu = (torch.relu, lambda Z: (Z > 0).float())
sigmoid = (torch.sigmoid, lambda Z: torch.sigmoid(Z) * (1 - torch.sigmoid(Z)))
```

We compare three networks: ReLU with the weights of Section 4.2, $\tau = 0.01$; ReLU with $\tau = 1$; and the sigmoid with $\tau^2 = 1/256$, so that $h\tau^2 = 1$ and only the slope of the

activation function changes the scale. The plot shows the mean square of the gradient at each layer; the top layer starts at 1.

```
runs = {'ReLU, N(0, 0.01²)': moments(*relu, 0.01),
        'ReLU, N(0, 1)': moments(*relu, 1.0),
        'sigmoid, N(0, 1/256)': moments(*sigmoid, 1 / 16)}
for name, (acts, grads) in runs.items():
    print(f'{name}: gradient mean square at layer 1: {grads[0]:.1e}')
d2l.plot(list(range(1, 11)), [grads for acts, grads in runs.values()],
         'layer', 'mean square of gradient', legend=list(runs),
         yscale='log', figsize=(4.5, 3))
```

```
ReLU, N(0, 0.01²): gradient mean square at layer 1: 1.0e-17
ReLU, N(0, 1): gradient mean square at layer 1: 8.2e+18
sigmoid, N(0, 1/256): gradient mean square at layer 1: 5.6e-12
```



On a logarithmic axis the curves are straight lines: every layer multiplies the mean square by the same factor. Over the nine steps from the top layer down to the first, the factors of Equation 4.21 predict $0.0128^9 \approx 10^{-17}$ for the small weights and $128^9 \approx 10^{19}$ for the large ones, and the measurement agrees with both. The independence assumption has passed its test, at least for random ReLU networks of this width. The sigmoid loses more than an order of magnitude per layer even with $h\tau^2 = 1$, and the mean square of its gradient at the first layer is about 5×10^{-12} of that at the top.

4.4.1.2 Vanishing Gradients

The two falling curves show the vanishing gradients that Section 4.1.4 met for the sigmoid: the weights of the first layers receive gradients many orders of magnitude smaller than those of the last layers. Any learning rate small enough to keep the last layers stable then leaves the first layers almost where they started. Small weights cause this, and so do activation functions with small slopes. Since the derivative of the sigmoid never exceeds $1/4$, we have $\mathbb{E}[\sigma'(z)^2] \leq 1/16$, and a sigmoid layer divides the mean square of the gradient by at least

16 even with $h\tau^2 = 1$. Could larger weights not compensate? Unfortunately not, because they push more pre-activations into the tails of the sigmoid, where its slope is close to zero (Bengio et al. 1994). The ReLU has slope 1 for every active unit, so only the scale of the weights sets its factor, which is one reason it replaced the sigmoid in hidden layers.

4.4.1.3 Exploding Gradients

The rising curve shows an *exploding gradient*. With a mean square 10^{19} times larger at the first layer than at the top, the gradient there is about 3×10^9 times larger in magnitude, and no single learning rate suits all layers: one small enough for the first layers leaves the last ones frozen, and one suited to the last layers makes the updates of the first ones overshoot. The forward pass grows by the same factor (Section 4.4.2.2). Once values exceed about 3.4×10^{38}, the largest number that 32-bit floating point can represent, they become infinite, and the loss becomes `nan`. A loss that is `nan` from the first step is often the sign of an initialization that is too large.

4.4.2 Parameter Initialization

4.4.2.1 Xavier Initialization

Which variance, then, keeps the factor of a layer at 1? Let us start with the simplest case, a layer without activation function, and ask first what it does to the signal in the forward pass. It has n_{in} inputs x_i and n_{out} outputs, and output j is

$$o_j = \sum_{i=1}^{n_{\text{in}}} x_i w_{ij}.$$

The weights w_{ij} are drawn independently from the same distribution, with mean zero and variance τ^2; the distribution need not be Gaussian. The inputs x_i have mean zero and variance γ^2, and they are independent of the weights and of each other. Then o_j has mean $\mathbb{E}[o_j] = \sum_i \mathbb{E}[x_i]\,\mathbb{E}[w_{ij}] = 0$ and variance

$$\begin{aligned}\text{Var}[o_j] = \mathbb{E}[o_j^2] &= \sum_{i=1}^{n_{\text{in}}} \sum_{i'=1}^{n_{\text{in}}} \mathbb{E}[x_i x_{i'} w_{ij} w_{i'j}] \\ &= \sum_{i=1}^{n_{\text{in}}} \mathbb{E}[x_i^2]\,\mathbb{E}[w_{ij}^2] = n_{\text{in}}\,\tau^2\gamma^2.\end{aligned}$$

In the second line, the cross terms with $i \neq i'$ vanish for the same reason as in the backward pass, and the remaining terms use $\mathbb{E}[w_{ij}^2] = \tau^2$ and $\mathbb{E}[x_i^2] = \gamma^2$, which hold because both have mean zero. The layer thus multiplies the variance of the signal by $n_{\text{in}}\tau^2$. To keep it unchanged, we need $n_{\text{in}}\tau^2 = 1$.

For the backward pass the work is already done. The sum behind Equation 4.21 runs over the $h = n_{\text{out}}$ outputs of the layer, and without activation function the slope is 1, so the mean square of the gradient is multiplied by $n_{\text{out}}\tau^2$. Keeping it unchanged needs $n_{\text{out}}\tau^2 = 1$. We cannot have both unless $n_{\text{in}} = n_{\text{out}}$. *Xavier initialization* (Glorot and Bengio 2010) settles for a compromise and satisfies their average,

$$\frac{1}{2}(n_{\text{in}} + n_{\text{out}})\,\tau^2 = 1, \qquad \text{that is,} \qquad \tau^2 = \frac{2}{n_{\text{in}} + n_{\text{out}}}.$$

For the first layer of the MLP of Section 4.2, with $n_{\text{in}} = 784$ and $n_{\text{out}} = 256$, this gives $\tau = \sqrt{2/1040} \approx 0.044$, four times the 0.01 used there. The weights may also be drawn uniformly. The uniform distribution on $[-a, a]$ has variance $a^2/3$, so Xavier initialization draws them from

$$U\left(-\sqrt{\frac{6}{n_{\text{in}} + n_{\text{out}}}}, \sqrt{\frac{6}{n_{\text{in}} + n_{\text{out}}}}\right). \tag{4.22}$$

The derivation assumed a layer without activation function. It also holds approximately for an activation function with slope 1 near zero, such as tanh, as long as the pre-activations stay small, and Glorot and Bengio derived it for tanh networks.

4.4.2.2 He Initialization

A ReLU is far from linear: it sets about half of its inputs to zero. To extend the argument, we note what it used. With independent weights of mean zero, the output o_j has mean zero and $\mathbb{E}[o_j^2] = n_{\text{in}}\tau^2\mathbb{E}[x_i^2]$ for *any* inputs, zero mean or not: the inputs enter only through their mean square $\mathbb{E}[x_i^2]$. So we track the mean square of the activations through the ReLU. If the pre-activations z are symmetric about zero, as they are when the weights are drawn from a symmetric distribution independently of the inputs, then $\text{ReLU}(z)^2$ equals z^2 for the positive half of them and 0 for the negative half, so the ReLU halves the mean square: $\mathbb{E}[\text{ReLU}(z)^2] = \frac{1}{2}\mathbb{E}[z^2]$.

Let $v^{(k)}$ be the mean square of the entries of $\mathbf{H}^{(k)}$, the activations of layer k. Its pre-activations have mean square $n_{\text{in}}\tau^2 v^{(k-1)}$, and the ReLU halves it:

$$v^{(k)} = \frac{1}{2}\, n_{\text{in}}\, \tau^2\, v^{(k-1)}. \tag{4.23}$$

The mean square stays constant from layer to layer if we double the weight variance of the linear case:

$$\tau^2 = \frac{2}{n_{\text{in}}}. \tag{4.24}$$

This is *He initialization* (He et al. 2015), also called Kaiming initialization, the standard choice for networks with ReLU activations. For the backward pass, Equation 4.21 with $\mathbb{E}[\phi'(z)^2] = 1/2$ gives the same condition with n_{out} in place of n_{in}, and for layers of equal width the two coincide. One weight layer has no ReLU after it, so the scale is off there by a single factor of 2, which does not compound with depth. For the first layer of the MLP of Section 4.2, He initialization gives $\tau = \sqrt{2/784} \approx 0.05$. The rule of thumb is Xavier for tanh and He for ReLU, and most libraries provide both as named initializers.

4.4.2.3 Default Initialization

A framework initializes the weights of a layer when we do not, and we can now judge its choice against the $2/n_{\text{in}}$ that a ReLU layer needs. None of the four frameworks defaults to it, so from the next section on our models set He initialization explicitly.

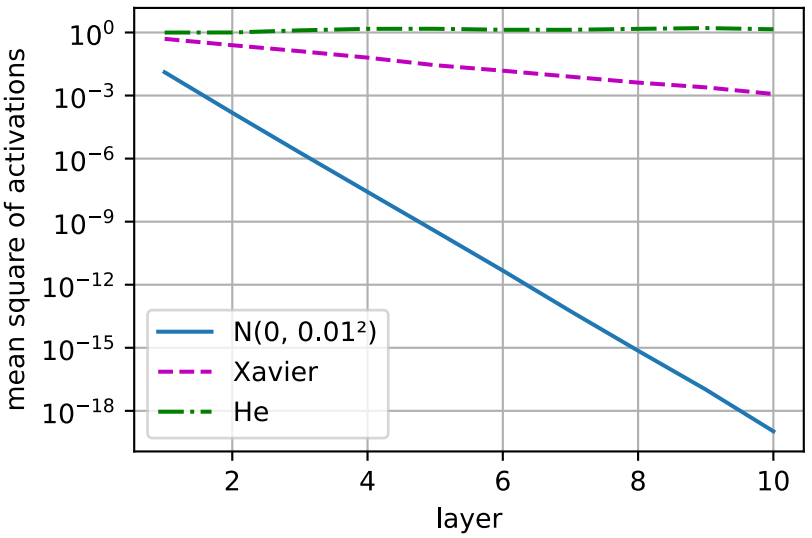
`nn.Linear` draws its weights and its biases uniformly from the interval $[-1/\sqrt{n_{\text{in}}}, 1/\sqrt{n_{\text{in}}}]$, so the weights have variance $1/(3n_{\text{in}})$, one sixth of what He initialization prescribes.

4.4.2.4 Activations and Gradients across Depth

We now repeat the measurement with ReLU and three initializations: the $\mathcal{N}(0, 0.01^2)$ of Section 4.2, Xavier, and He. With $n_{\text{in}} = n_{\text{out}} = 256$, Xavier gives $\tau^2 = 1/256$ and He gives $\tau^2 = 2/256$. This time we plot the mean square of the activations and print its value at the last layer, together with the mean square of the gradient at the first.

```
runs = {'N(0, 0.01²)': moments(*relu, 0.01),
        'Xavier': moments(*relu, (1 / 256) ** 0.5),
        'He': moments(*relu, (2 / 256) ** 0.5)}
for name, (acts, grads) in runs.items():
    print(f'{name}: activations at layer 10: {acts[-1]:.1e}, '
          f'gradient at layer 1: {grads[0]:.1e}')
d2l.plot(list(range(1, 11)), [acts for acts, grads in runs.values()],
         'layer', 'mean square of activations', legend=list(runs),
         yscale='log', figsize=(4.5, 3))
```

```
N(0, 0.01²): activations at layer 10: 1.1e-19, gradient at layer 1: 9.2e-18
Xavier: activations at layer 10: 1.2e-03, gradient at layer 1: 2.1e-03
He: activations at layer 10: 1.4e+00, gradient at layer 1: 1.3e+00
```



He initialization keeps the mean squares of the activations and of the gradients within a factor of about 3 of 1 through all ten layers. The curve is not exactly flat, because the factor of each layer fluctuates around 1 with the random weights and the fluctuations compound.

Xavier initialization loses the factor $1/2$ of the ReLU at every layer, in both directions. The predictions are $2^{-10} \approx 10^{-3}$ for the activations at the last layer and $2^{-9} \approx 2 \times 10^{-3}$ for the gradient at the first; the measurements lie within a factor of 3 of them. With the weights of Section 4.2, the mean square of the activations at the last layer is about 10^{-19}. To every input alike, the last hidden layer answers with entries of size about 3×10^{-10}.

4.4.2.5 Training a Deep Network

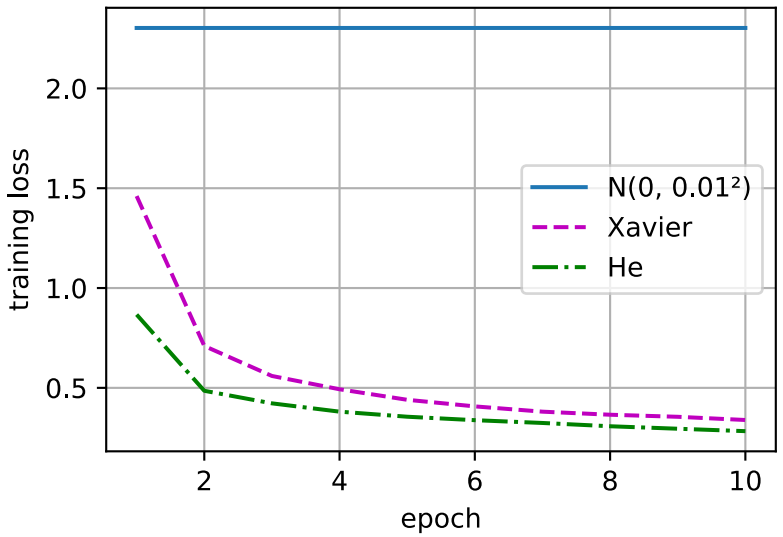
So far we have measured networks that were never trained. Do the measurements predict what training does? We train the network of ten hidden ReLU layers of width 256 on Fashion-MNIST with each of the three initializations, for ten epochs of minibatch SGD with minibatches of 256, and plot the training loss. The learning rate is 0.05, half that of Section 4.2, because at 0.1 this deep network now and then diverges after a few epochs; we come back to the reason below. `DeepMLP` builds the network from the framework's layers. It draws the weights of a layer with n_{in} inputs and n_{out} outputs with the standard deviation `std(n_in, n_out)` and sets the biases to 0.

```
class DeepMLP(d2l.Classifier):
    """A deep MLP whose weights have the standard deviation std(n_in, n_out)."""
    def __init__(self, std, lr, depth=10, width=256):
        super().__init__()
        self.save_hyperparameters(ignore=['std'])
        sizes = [784] + [width] * depth + [10]
        layers = [nn.Flatten()]
        for n_in, n_out in zip(sizes[:-1], sizes[1:]):
            layer = nn.Linear(n_in, n_out)
            nn.init.normal_(layer.weight, std=std(n_in, n_out))
            nn.init.zeros_(layer.bias)
            layers += [layer, nn.ReLU()]
        self.net = nn.Sequential(*layers[:-1])  # No ReLU after the logits
```

The three initializations differ only in the function `std`. For each we print the validation accuracy after the last epoch.

```
inits = {'N(0, 0.01²)': lambda n_in, n_out: 0.01,
         'Xavier': lambda n_in, n_out: (2 / (n_in + n_out)) ** 0.5,
         'He': lambda n_in, n_out: (2 / n_in) ** 0.5}
data = d2l.FashionMNIST(batch_size=256)
losses = {}
for name, std in inits.items():
    trainer = d2l.Trainer(max_epochs=10, num_gpus=1)
    trainer.board.display = False
    trainer.fit(DeepMLP(std, lr=0.05), data)
    losses[name] = trainer.history['train_loss']
    print(f'{name}: validation accuracy {trainer.history["val_acc"][-1]:.2f}')
d2l.plot(list(range(1, 11)), list(losses.values()), 'epoch', 'training loss',
         legend=list(losses), figsize=(4.5, 3))
```

```
N(0, 0.01²): validation accuracy 0.10
Xavier: validation accuracy 0.86
He: validation accuracy 0.85
```



The measurements predicted the outcome. With weights of standard deviation 0.01 the network learns nothing. Its training loss sits at $\log 10 \approx 2.30$ for all ten epochs, the loss of the untrained classifier, and its validation accuracy is the 10% of guessing. A gradient whose mean square at the first layer is 10^{17} times smaller than at the top is far too small to move the weights there, and the last hidden layer passes on nearly nothing for the output layer to learn from. Xavier and He initialization both train. Xavier initialization, which loses a factor of 2 per layer, starts more slowly, with a training loss of about 1.5 after the first epoch against about 0.85 for He initialization, and it stays a little behind. After ten epochs both networks reach a validation accuracy of about 85%, no better than the MLP with one hidden layer. The nine additional layers were not expected to help on this problem. What the experiment shows is that they can be trained, and that the initial scale decides whether they are.

A good initialization is a statement about the untrained network only. Our derivations assumed that the weights are independent of each other and of the inputs, which holds when they are drawn. Training changes the weights and breaks the assumption, and nothing then keeps the factor of each layer near 1. This is how the deep network can diverge at the larger learning rate although it started at the right scale. Normalization layers (Section 7.3) and residual connections, introduced with deeper architectures later in the book, keep the scale under control during training as well.

4.4.3 Symmetry among Hidden Units

The variance argument fixes the scale of the initial weights. It does not say why they should be random. After all, a constant would have the same mean square. Consider an MLP with one hidden layer and a single output, and suppose that every weight of the hidden layer equals the same constant c and every weight of the output layer the same constant c'. Every hidden unit then computes the same function of the input. In the backward pass, every hidden unit receives the same gradient from the output, and the weights into each unit receive the same gradient. After an update of minibatch stochastic gradient descent, the units are still identical, and by the same argument after every later update. The h hidden units behave like a single unit: 256 of them would be one unit, stored 256 times. The hidden units of

an MLP are interchangeable: permuting them, together with their incoming and outgoing weights, does not change the function that the network computes. Gradient descent cannot break this symmetry if the initialization does not.

We check this with $d = 20$ inputs, $h = 16$ hidden ReLU units, a minibatch of 64 random examples with random targets, and the squared loss. The constant $c = \sqrt{2/d}$ gives the pre-activations the same mean square as He initialization. We compute the gradient with respect to the weights of the hidden layer and print the largest difference between the gradients of two hidden units, that is, between two columns of the gradient, for the constant weights and for random weights at the same scale.

```
def hidden_grad(W1, W2, X, y):
    """Gradient of the squared loss with respect to the hidden weights."""
    W1 = W1.clone().requires_grad_(True)
    L = ((torch.relu(X @ W1) @ W2 - y) ** 2).mean() / 2
    L.backward()
    return W1.grad

d, h = 20, 16
X, y = torch.randn(64, d), torch.randn(64, 1)
c, c_out = (2 / d) ** 0.5, (1 / h) ** 0.5
inits = {'constant': (torch.full((d, h), c), torch.full((h, 1), c_out)),
         'random': (c * torch.randn(d, h), c_out * torch.randn(h, 1))}
for name, (W1, W2) in inits.items():
    G = hidden_grad(W1, W2, X, y)
    print(f'{name}: largest difference between two hidden units '
          f'{float((G - G[:, :1]).abs().max()):.2f}')
```

```
constant: largest difference between two hidden units 0.00
random: largest difference between two hidden units 0.28
```

With constant weights, all 16 columns of the gradient are identical, as the argument said they must be. With random weights, the units receive different gradients from the first step on. The biases can start at zero, since the weights already differ between units.

A network whose gradients arrive intact is not yet a network that predicts well on new data. The next section asks why deep networks, with many more parameters than training examples, generalize at all.

4.4.4 Summary

The backward pass multiplies the gradient by one matrix of slopes and one transposed weight matrix per layer, so the gradient that reaches the first layers of a deep network is a long product. Each ReLU layer of width h with weights of variance τ^2 multiplies its mean square by about $h\tau^2/2$, and the mean square of the activations by the same factor on the way up. Unless this factor is close to 1, ten layers suffice to change both by many orders of magnitude. Saturating activation functions such as the sigmoid shrink it even when the weights are scaled correctly. Xavier initialization, $\tau^2 = 2/(n_{\text{in}} + n_{\text{out}})$, keeps the variance constant through linear layers, and He initialization, $\tau^2 = 2/n_{\text{in}}$, compensates for the ReLU, which halves the mean square. Trained on Fashion-MNIST, the network with ten hidden layers never leaves its initial loss with weights of standard deviation 0.01, and learns from the first epoch with He initialization. The initial weights must also be random, because hidden units

that start identical receive identical gradients and stay identical.

4.4.5 Exercises

1. **Spectral norm bounds.** For a single example, written as a column vector, the step Equation 4.19 multiplies the gradient $\mathbf{g}'$ above layer k by the matrix $\mathbf{M}^{(k)} = \mathbf{W}^{(k)} \operatorname{diag}(\phi'(\mathbf{z}^{(k)}))$, where $\operatorname{diag}(\cdot)$ places the entries of a vector on the diagonal of an otherwise zero matrix, so the gradient in Equation 4.20 is a product of $K-1-k$ such matrices times the gradient at the top. The spectral norm (the largest singular value) is submultiplicative: $\|\mathbf{AB}\|_2 \le \|\mathbf{A}\|_2 \|\mathbf{B}\|_2$.
   1. Prove submultiplicativity from the definition $\|\mathbf{A}\|_2 = \max_{\mathbf{x}\neq 0} \|\mathbf{Ax}\|/\|\mathbf{x}\|$.
   2. Use it to bound the norm of the gradient in terms of the norms $\|\mathbf{M}^{(k)}\|_2$ of the factors. Which condition on these norms keeps the bound from growing geometrically with depth?
   3. This bound is only an upper bound. Explain why the condition of the previous part rules out exploding gradients but not vanishing ones, and give two 2×2 matrices that satisfy the condition with equality but whose product has spectral norm strictly smaller than one.

2. **Other symmetries.** Permuting the hidden units is not the only change of parameters that leaves the function of an MLP unchanged. Consider a one-hidden-layer MLP in which hidden unit j has incoming weights $\mathbf{w}_j$, bias b_j, and outgoing weights $\mathbf{v}_j$.
   1. Prove that $\operatorname{ReLU}(\alpha z) = \alpha \operatorname{ReLU}(z)$ for $\alpha > 0$. Use this identity to show that replacing $(\mathbf{w}_j, b_j, \mathbf{v}_j)$ by $(\alpha\mathbf{w}_j, \alpha b_j, \mathbf{v}_j/\alpha)$ leaves the function of a ReLU network unchanged.
   2. Find a transformation of the parameters of a single hidden unit that leaves the function of a tanh network unchanged.
   3. Identical hidden units are a fixed point of the permutation, and gradient descent started at that fixed point stays there. Find the parameters of a unit that the transformation of the previous part leaves unchanged. Then show that if all weights and biases of a tanh network are initialized to zero, gradient descent changes only the output bias, at every step.

   Adapted from Simon Prince, Understanding Deep Learning, Problems 3.5, 4.3, and 7.15.

3. **Fan-in versus averaged fans.** Following He et al. (2015), consider a linear network with K layers in which layer k maps n_{k-1} inputs to n_k outputs and draws its weights with variance τ_k^2. By the computation of Section 4.4.2.1, layer k multiplies the variance of the forward signal by $n_{k-1}\tau_k^2$ and that of the backward signal by $n_k\tau_k^2$.
   1. Set $\tau_k^2 = 1/n_{k-1}$ in every layer. Show that the forward variance is preserved and that the backward variance changes by the total factor n_K/n_0 at any depth.
   2. Let the widths alternate as $100, 400, 100, 400, \ldots$, and use Xavier initialization in every layer. Compute the factor by which one $100 \to 400 \to 100$ pair of layers multiplies the forward variance, and the factor after 50 such pairs.
   3. Which of the two rules keeps both variances within a depth-independent factor of their initial values for every sequence of widths?

4. code **Linear networks.** Rerun the measurement of Section 4.4.2.4 without activation function, by passing the identity and its derivative to `moments`, with the weight scales of $\mathcal{N}(0, 0.01^2)$, Xavier, and He.

1. Compute the factor $n_{\text{in}}\tau^2$ per layer for each of the three scales, and check the factors against the slopes of the measured curves.
2. Compare the ranking of the three initializations with the ranking for the ReLU network and explain the difference.

5. code **Layerwise adaptive scaling.** ⊠ Write $\mathbf{g}^{(k)} = \partial L/\partial \mathbf{W}^{(k)}$ for the gradient of the loss L with respect to the weights of layer k. When the gradients of different layers differ by orders of magnitude, a natural remedy is to give each layer a step proportional to its own weights (You et al. 2017). It replaces the SGD update $\mathbf{W}^{(k)} \leftarrow \mathbf{W}^{(k)} - \eta\,\mathbf{g}^{(k)}$ by

$$\mathbf{W}^{(k)} \leftarrow \mathbf{W}^{(k)} - \eta\,\frac{\|\mathbf{W}^{(k)}\|_{\text{F}}}{\|\mathbf{g}^{(k)}\|_{\text{F}}}\,\mathbf{g}^{(k)}. \tag{4.25}$$

The scaled update has norm $\eta\,\|\mathbf{W}^{(k)}\|_{\text{F}}$: every layer moves by the same fraction η of its weight norm, however large or small its gradient. An exploding gradient therefore changes only the direction of the update, not its size.

1. Build a three-layer linear network of width 100 with $\mathcal{N}(0,1)$ weights. Draw one fixed minibatch of 256 inputs and 256 targets in $\mathbb{R}^{100}$ from $\mathcal{N}(\mathbf{0},\mathbf{I})$, and let L be the mean squared error on this minibatch. Train with plain SGD at $\eta = 0.1$, record L and $\|\mathbf{g}^{(1)}\|_{\text{F}}$ over the first ten steps, and show that both diverge.
2. Replace the update by the scaled one, with the same η, and repeat the ten steps. Does the loss decrease? Relate the outcome to the scale of the network's outputs at this initialization.

6. **Variance under ReLU.** ⊠ The derivation in Section 4.4.2.2 tracks the mean square $\mathbb{E}[\text{ReLU}(z)^2]$ rather than the variance, because the next layer's pre-activation $o = \sum_i x_i w_i$, with inputs $x_i = \text{ReLU}(z_i)$ and independent zero-mean weights has $\text{Var}[o] = n_{\text{in}}\tau^2\mathbb{E}[x_i^2]$ for any mean of x_i.

1. For $z \sim \mathcal{N}(0,\nu^2)$, compute $\mathbb{E}[\text{ReLU}(z)]$ and $\text{Var}[\text{ReLU}(z)]$, and show that the variance equals $\left(\frac{1}{2} - \frac{1}{2\pi}\right)\nu^2$ rather than $\frac{1}{2}\nu^2$.
2. A tempting shortcut substitutes $\text{Var}[x_i]$ for $\mathbb{E}[x_i^2]$ in the Xavier computation. Show that this shortcut prescribes $\tau^2 = 2/(n_{\text{in}}(1-1/\pi))$ instead of $2/n_{\text{in}}$. Compute the factor by which each ReLU layer initialized this way actually multiplies the mean square, and the total factor over fifty layers.

7. code **Tanh networks.** ⊠ Repeat the measurement of Section 4.4.2.4 with tanh in place of ReLU, with Xavier initialization and fifty layers. A layer then maps the mean square m of its input to $m' = \mathbb{E}[\tanh(z)^2]$, where z is approximately $\mathcal{N}(0,m)$.

1. Use $\tanh(z)^2 \approx z^2 - \frac{2}{3}z^4$ for small z to show that $m' \approx m - 2m^2$. Over many layers, does the mean square stay constant, decay geometrically, or decay more slowly? Show that the mean square at layer k is about $1/(2k)$ for large k.
2. Check the prediction by plotting the measured mean square against the layer on logarithmic axes. What happens to the gradient?

4.5 Generalization in Deep Learning

In Section 2.6 we saw what usually happens when a model has as many parameters as training examples. A polynomial with one weight per training example passed through all 20 of its training points and missed new points by orders of magnitude. The MLP of Section 4.2 would seem to be in greater danger still. It has 203,530 parameters, about 3.4 for each of the 60,000 Fashion-MNIST training images, plenty to memorize them. After 30 epochs it had not: its accuracy of about 89% on the training images was only a few points above the 87% on validation images. That run was short, however, and proves little. What happens if we train such a network until it fits every training label? We will find that it still generalizes, so something other than the number of parameters must be responsible.

We need to know what that is, because the width of a network, the length of its training, and its regularization are all chosen for the sake of generalization. The answer is not completely known, so this section measures. We first show that the MLP can fit labels that carry no information, so its capacity alone cannot explain why it generalizes. Then we follow the test error of a simpler network as its width grows past the number of training examples, and see it fall a second time. Both observations point to the training algorithm: among the many parameter settings that fit the data, gradient descent finds particular ones, and we work out which in two cases that can be solved exactly. Finally, the order in which gradient descent fits the examples turns stopping early into a regularizer. We stay with experiments and with exact results about simple models; theories that bound the test error of deep networks directly are still being developed.

4.5.1 Memorization and Capacity

4.5.1.1 Random Labels

The direct test of capacity is to train the network on labels drawn at random, independently of the images, as Zhang et al. (2021) did for image classifiers. There is no pattern to learn from such labels, so a network that fits them has memorized them: its training accuracy reaches 100% while its test accuracy stays at the 10% of guessing. To keep the experiment fast, we train on 1,000 Fashion-MNIST images, which gives the MLP about 200 parameters per image.

We draw 2,000 training images at random, 1,000 to train on and 1,000 to hold out for validation later, and take the first 2,000 test images. All of them are flattened into NumPy arrays of 784 pixels.

```
data = d2l.FashionMNIST(batch_size=2000)
X, y = next(iter(data.train_dataloader()))          # 2,000 random images
X_test, y_test = next(iter(data.val_dataloader()))  # the first 2,000
X, y, X_test, y_test = (d2l.numpy(a) for a in (X, y, X_test, y_test))
X, X_test = X.reshape(2000, -1), X_test.reshape(2000, -1)
X_train, y_train, X_val, y_val = X[:1000], y[:1000], X[1000:], y[1000:]
```

Two small helpers run the experiments. `ArrayData` serves arrays as minibatches. `Tracker` is a `Trainer` that, after every epoch, measures the accuracy of the model on each of a set of named datasets and appends it to the list `accuracy[name]`.

```
class ArrayData(d2l.DataModule):
    """Minibatches of the arrays X and y; training batches are shuffled."""
    def __init__(self, X, y, batch_size=64):
        super().__init__()
        self.save_hyperparameters()

    def get_dataloader(self, train):
        return self.get_tensorloader(
            (d2l.tensor(self.X), d2l.tensor(self.y)), train)

class Tracker(d2l.Trainer):
    """A Trainer that records the accuracy on named datasets every epoch."""
    def __init__(self, max_epochs, datasets):
        super().__init__(max_epochs, num_gpus=1)
        self.datasets = datasets
        self.accuracy = {name: [] for name in datasets}
        self.board.display = False  # We plot the accuracies ourselves

    def validate(self):
        for name, dataset in self.datasets.items():
            self.val_dataloader = dataset.val_dataloader()
            self.accuracy[name].append(super().validate()['acc'])
        return {}
```

The model is the MLP of Section 4.2, one hidden layer of 256 ReLUs, with one change. It draws its weights by the He initialization of Section 4.4.2.2 instead of with the fixed standard deviation of 0.01, which made the hidden layer learn slowly at first. Its biases start at zero, and its inputs are already flattened.

```
class MLP(d2l.Classifier):
    def __init__(self, num_hiddens, lr):
        super().__init__()
        self.save_hyperparameters()
        self.net = nn.Sequential(nn.LazyLinear(num_hiddens), nn.ReLU(),
                                 nn.LazyLinear(10))

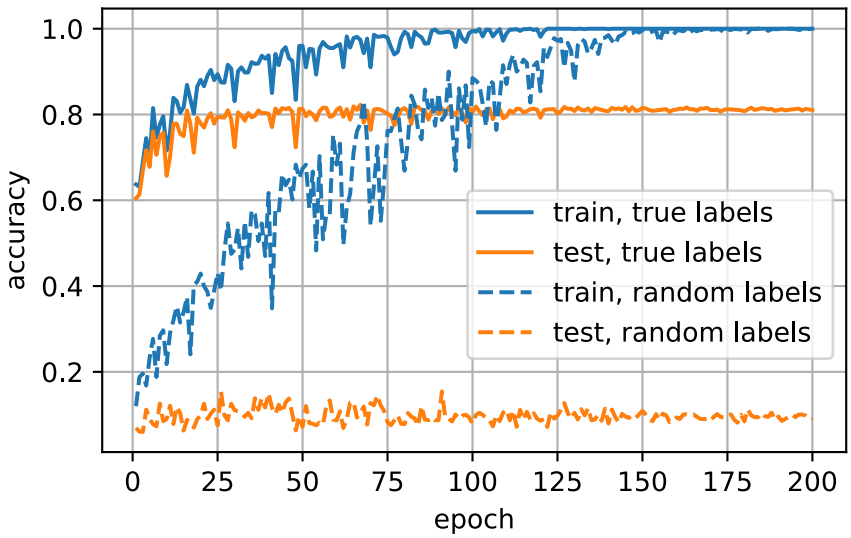
    def weight_init(self, module):
        if isinstance(module, nn.Linear):
            nn.init.kaiming_normal_(module.weight, nonlinearity='relu')
            nn.init.zeros_(module.bias)
```

`corrupt` replaces each label, with probability `fraction`, by a class drawn uniformly at random; with `fraction=1` no label depends on its image. Does the network fit true and random labels alike? We train it on both for 200 epochs of minibatch SGD, with learning rate 0.1 and minibatches of 64 images, and record the accuracy on the training images and on the test images.

```
def corrupt(y, fraction, rng):
    """Replace each label, with probability fraction, by a random class."""
    redraw = rng.random(len(y)) < fraction
    y = y.copy()
```

```
    y[redraw] = rng.integers(0, 10, redraw.sum())
    return y

rng = np.random.default_rng()
curves = []
for fraction in (0, 1):
    labels = corrupt(y_train, fraction, rng)
    trainer = Tracker(max_epochs=200, datasets={
        'train': ArrayData(X_train, labels, 1000),
        'test': ArrayData(X_test, y_test, 1000)})
    trainer.fit(MLP(num_hiddens=256, lr=0.1), ArrayData(X_train, labels))
    curves += [trainer.accuracy['train'], trainer.accuracy['test']]
d2l.plot(list(range(1, 201)), curves, 'epoch', 'accuracy',
         fmts=('C0-', 'C1-', 'C0--', 'C1--'), figsize=(5, 3))
d2l.plt.legend(['train, true labels', 'test, true labels',
                'train, random labels', 'test, random labels'],
               loc='center right', bbox_to_anchor=(1, 0.4));
```



With true labels, training accuracy passes 99% after 60 to 90 epochs, and test accuracy settles at about 80% to 83%. A network trained until it fits its training labels still gets about four of five new images right. With random labels the network needs longer, 110 to 150 epochs to pass 99%, but it fits them too, while test accuracy stays where guessing would put it. The network, the optimizer, and the images are the same in both runs; only the labels differ.

For these 1,000 images, MLPs of this size come close to what Section 2.6 called a class too flexible for a perfect fit to mean anything. Whatever lets the network generalize from the true labels, it is not a lack of capacity to memorize.

4.5.1.2 Capacity Bounds

Learning theory turns capacity into guarantees. In Chapter 3.6 we met the VC bound: with probability at least $1 - \delta$ over the draw of n training examples, every classifier f in a class of VC dimension v satisfies

$$\epsilon(f) \le \epsilon_{\mathcal{S}}(f) + c\sqrt{\frac{v + \log(1/\delta)}{n}},$$

where $\epsilon(f)$ is the population error, $\epsilon_{\mathcal{S}}(f)$ the training error, and c a constant. The bound guarantees that the training error carries over to new data once n is large compared with v. For a linear classifier, v is one more than the number of features, the number of its parameters. If we estimate v the same way for the MLP trained on all 60,000 images, the square root alone is $\sqrt{203{,}530/60{,}000} \approx 1.8$. An error never exceeds 1, so unless the constant c is below about 0.55, the bound guarantees nothing.

Could a smaller constant or a sharper estimate of v rescue it? Unfortunately not, and the random labels tell us why. A bound whose second term depends only on the class, n, and δ holds for every distribution of the data, including images with random labels. On our 1,000 images with such labels the network had training error 0, while its population error is 90%, so for this n the second term is at least 0.9. The very same term applies when the labels are true, where it guarantees next to nothing (exercise 1).

4.5.1.3 Inductive Bias

If the class does not decide, the choice among the functions that fit the training data does. A learner that fits n labeled examples must still label every other input, and the training data cannot settle how. Averaged over all ways in which the unseen inputs could be labeled, every learner has the same expected error, a result known as the *no free lunch theorem* (Wolpert 1996). A learner generalizes on a particular problem when its preferences among the functions that fit match the way the problem labels new inputs. These preferences are its *inductive bias*. Different learners make different bets. The *1-nearest-neighbor* rule, for one, gives an input the label of the closest training input.

Learner	Inductive bias
Linear classifier	classes are separated by hyperplanes
Polynomial fit with weight decay	small weights, hence slowly varying curves
1-nearest neighbor	nearby inputs have the same label
MLP trained by gradient descent	not fully known; the rest of this section

The nearest-neighbor rule is instructive. It labels each training input correctly by construction, yet under mild conditions, as the training set grows, its error tends to at most twice the smallest error any classifier can achieve (Cover and Hart 1967). Fitting the data exactly is thus compatible with generalizing. The error is decided by the labels the learner assigns between the training points.

4.5.2 Double Descent

4.5.2.1 The Interpolation Threshold

The polynomial study of Section 2.6 stopped at degree 19, where the fit was exact and the test error at its worst, because beyond that degree the least-squares problem has infinitely many exact solutions. A network with more parameters than examples lives beyond that point. Let us therefore continue past it, with a model whose solutions we can compute.

To this end we simplify the MLP rather drastically. Freeze the hidden layer at its random initialization and train only the output layer, which gives what is called a *random-features* model (Rahimi and Recht 2007). We also drop the biases and, instead of cross-entropy, use the squared loss on one-hot labels, which turns training into linear least squares. Real networks change their hidden features during training; we give this up for a model we can solve. Let $\mathbf{H} = \mathrm{ReLU}(\mathbf{X}\mathbf{W}^{(1)}) \in \mathbb{R}^{n\times h}$ be the hidden representation of the n training images with h hidden units, and $\mathbf{Y} \in \mathbb{R}^{n\times 10}$ their one-hot labels. Fitting chooses the output weights $\mathbf{W}^{(2)} \in \mathbb{R}^{h\times 10}$ to minimize the squared error summed over all entries, $\|\mathbf{H}\mathbf{W}^{(2)} - \mathbf{Y}\|_F^2$.

Each output is a separate least-squares problem with n equations and h unknowns. The MLP of Section 4.5.1.1 could fit 1,000 random labels with 256 hidden units because its hidden layer was trained too. With that layer frozen, an exact fit of n labels needs $h \geq n$. For $h < n$ the least-squares problem has a unique solution, which usually misses some labels. At $h = n$ the matrix $\mathbf{H}$ is square and, like almost every random square matrix, invertible, so exactly one solution reproduces every label. For $h > n$ there are infinitely many solutions that reproduce every label, and we take the one of smallest norm. `np.linalg.lstsq` returns this solution in all three cases. The width $h = n$, at which an exact fit first becomes possible, is the *interpolation threshold*.

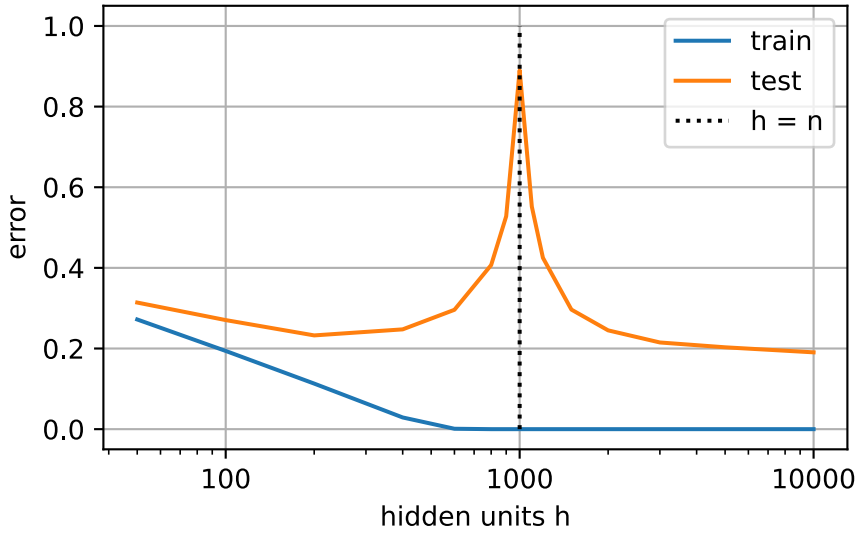
We compute the hidden representation once for 10,000 hidden units; the model with h units uses its first h columns. Before that we subtract the mean training image from every image. Since all pixels are nonnegative, a unit could otherwise be active on only a few images, or on none, and such nearly empty columns can make $\mathbf{H}$ lose rank. With centered images, every unit is active on a large share of them.

```
W1 = rng.normal(0, np.sqrt(2 / 784), (784, 10000))  # He-initialized, frozen
mean = X_train.mean(axis=0)                         # The mean training image
H = np.maximum((X_train - mean) @ W1, 0)
H_test = np.maximum((X_test - mean) @ W1, 0)
Y = np.eye(10)[y_train]                             # One-hot labels

def error(H, W2, y):
    """Fraction of examples whose largest output is not their label."""
    return (np.argmax(H @ W2, axis=1) != y).mean()
```

How does the error depend on the width h? We fit the output layer for widths from 50 to 10,000 and record the smallest singular value of each feature matrix, which `lstsq` returns, and the norm of the solution.

```
widths = [50, 100, 200, 400, 600, 800, 900, 1000, 1100, 1200, 1500, 2000,
          3000, 5000, 10000]
train_err, test_err, s_min, norm = [], [], {}, {}
for h in widths:
    W2, _, _, s = np.linalg.lstsq(H[:, :h], Y, rcond=None)
    train_err.append(error(H[:, :h], W2, y_train))
    test_err.append(error(H_test[:, :h], W2, y_test))
    s_min[h], norm[h] = s.min(), np.linalg.norm(W2)
d2l.plot([widths, widths, [1000, 1000]], [train_err, test_err, [0, 1]],
         'hidden units h', 'error', legend=['train', 'test', 'h = n'],
         xscale='log', fmts=('C0-', 'C1-', 'k:'), figsize=(5, 3))
```



The left part of the curve is the U of Section 2.6. Test error falls from about 30% at 50 hidden units to about 23% at a few hundred units, and then rises. At the interpolation threshold $h = n = 1{,}000$ it peaks at about 90%. The model now reproduces every training label, and on new images it does no better than a ten-sided die. Beyond the threshold every model fits all of its training labels, yet the test error falls again. At 10,000 hidden units it is back at about a fifth, a few points below the best width before the threshold. This shape is called *double descent* (Belkin et al. 2019).

The training curve counts misclassified images, so it reaches zero already at 600 to 800 hidden units, although the squared error vanishes only from $h = n$ on. Past the threshold, the training error cannot tell the models apart; the test error can.

4.5.2.2 Conditioning at the Threshold

Why does the error peak where the number of hidden units equals the number of examples? At $h = n$ the feature matrix $\mathbf{H}$ is square, and fitting must solve $\mathbf{H}\mathbf{W}^{(2)} = \mathbf{Y}$ exactly. A random square matrix is almost always invertible, but in some direction it is nearly singular: its smallest singular value $s_{\min}$ is close to zero, and its inverse stretches that direction by $1/s_{\min}$. To reproduce the labels, the solution needs a large component along it, and predictions on new images inherit that component. With fewer hidden units, least squares need not fit every label; with more, many solutions fit, and the one of smallest norm avoids the nearly singular direction.

```
for h in (200, 900, 1000, 1100, 10000):
    print(f'h = {h:5d}: smallest singular value {s_min[h]:.1e}, '
          f'norm of W2 {norm[h]:6.1f}')
```

```
h =   200: smallest singular value 1.7e+00, norm of W2    3.7
h =   900: smallest singular value 1.2e-01, norm of W2   22.0
h =  1000: smallest singular value 1.7e-05, norm of W2 28332.4
h =  1100: smallest singular value 1.0e-01, norm of W2   23.3
h = 10000: smallest singular value 2.0e+00, norm of W2    2.4
```

At the threshold the smallest singular value is tens to thousands of times smaller than at 1,100 hidden units, depending on the draw of the features. The norm of the solution jumps from about 20 at 900 units to several hundred or more, sometimes thousands, against about 20 at 1,100 units and 2 at 10,000. Beyond the threshold, the norm can only shrink as units are added: an exact fit with h hidden units remains an exact fit with $h + 1$ (give the new unit zero weight), so the smallest norm among the exact fits cannot grow. The mathematics appendix analyzes this mechanism in a smaller example (Section 36.10.3.2).

The threshold moves with the number of examples, which has a surprising consequence. With 1,000 hidden units, how does the test error change if we train on half as many images?

```
for n in (500, 1000):
    W2 = np.linalg.lstsq(H[:n, :1000], Y[:n], rcond=None)[0]
    test_error = error(H_test[:, :1000], W2, y_test)
    print(f'{n} training images: test error {test_error:.2f}')
```

```
500 training images: test error 0.29
1000 training images: test error 0.89
```

Doubling the training set from 500 to 1,000 images raises the test error from under a third to about 90%, because the larger set puts the model onto its interpolation threshold. More data can hurt.

Nakkiran et al. (2021) observed the same peaks in deep networks, as the width, the number of training epochs, or the number of examples varies, each time where training can just barely fit the training set. That is an observation. Our explanation by a nearly singular matrix holds for the random-features model only.

4.5.3 Implicit Bias of Gradient Descent

The second descent depended on taking the smallest solution among the many that fit, a choice we made by hand when we called `lstsq`. Nothing in the loss asks for it: every exact fit has zero training loss. Which fit does gradient descent find? For two models the answer is known exactly. Such a preference of a training algorithm is called its *implicit bias*.

4.5.3.1 Minimum-Norm Interpolation

We begin with the random-features model of the double-descent experiment. Its loss is quadratic, so the fit that gradient descent reaches can be written down. Consider one output of this model: weights $\mathbf{w} \in \mathbb{R}^h$, the loss $L(\mathbf{w}) = \frac{1}{2}\|\mathbf{H}\mathbf{w} - \mathbf{y}\|^2$ with $h > n$, and a matrix $\mathbf{H}$ whose n rows are linearly independent. The exact fit of smallest norm is

$$\mathbf{w}^\star = \mathbf{H}^\top(\mathbf{H}\mathbf{H}^\top)^{-1}\mathbf{y}.$$

It is a combination of the rows of $\mathbf{H}$, and it reproduces the labels, since $\mathbf{H}\mathbf{w}^\star = \mathbf{y}$.

Proposition. Gradient descent on L started at $\mathbf{w}_0 = \mathbf{0}$, with a learning rate small enough for it to converge, converges to $\mathbf{w}^\star$.

Proof. The gradient $\mathbf{H}^\top(\mathbf{H}\mathbf{w} - \mathbf{y})$ is a combination of the rows of $\mathbf{H}$. Starting from zero, every iterate is therefore a combination of the rows, $\mathbf{w}_t = \mathbf{H}^\top\mathbf{a}_t$ for some $\mathbf{a}_t \in \mathbb{R}^n$, and so is the limit. At the limit the gradient vanishes, so for this convex loss the limit is a minimizer.

Since the rows of $\mathbf{H}$ are independent, exact fits exist, so the limit fits exactly: $\mathbf{H}\mathbf{H}^\top\mathbf{a} = \mathbf{y}$, which gives $\mathbf{a} = (\mathbf{H}\mathbf{H}^\top)^{-1}\mathbf{y}$ and the limit $\mathbf{w}^\star$. It remains to see that no exact fit is shorter. Every exact fit is $\mathbf{w}^\star + \mathbf{v}$ with $\mathbf{H}\mathbf{v} = \mathbf{0}$. Such a $\mathbf{v}$ is orthogonal to every row of $\mathbf{H}$, hence to $\mathbf{w}^\star$, so $\|\mathbf{w}^\star + \mathbf{v}\|^2 = \|\mathbf{w}^\star\|^2 + \|\mathbf{v}\|^2$. ■

Figure 4.10 shows the argument for one equation in two unknowns. Every step moves perpendicular to the line of exact fits, so the component of a start $\mathbf{w}_0$ along the line never changes. Only a start without such a component ends at the point closest to the origin.

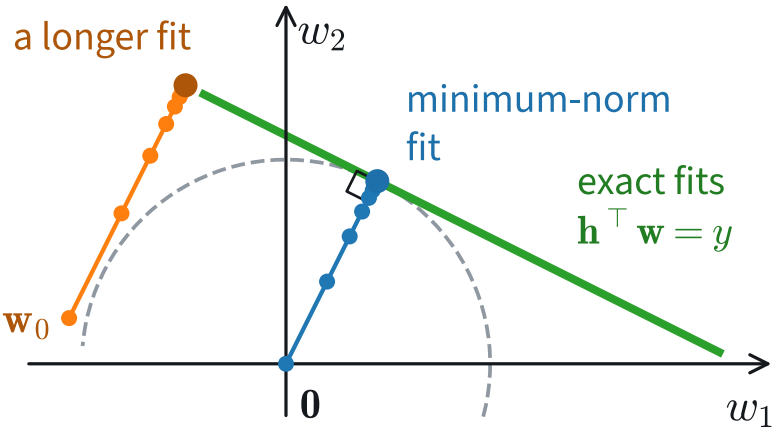


Figure 4.10: Gradient descent on least squares with one equation $\mathbf{h}^\top\mathbf{w} = y$ in two unknowns. The exact fits form a line (green). Each gradient is a multiple of $\mathbf{h}$, so each step moves perpendicular to that line. Started at $\mathbf{0}$ (blue), gradient descent stops at the point of the line closest to the origin, where the dashed circle touches it. Started at $\mathbf{w}_0$ (orange), it stops at a longer exact fit.

We check the proposition on a random problem with 20 equations and 100 unknowns, starting once at zero and once at a random point.

```
H_small = rng.normal(size=(20, 100))
y_small = rng.normal(size=20)
w_star = np.linalg.lstsq(H_small, y_small, rcond=None)[0]
lr = 1 / np.linalg.norm(H_small, 2) ** 2  # Below 2 / largest curvature
for name, w in (('zero', np.zeros(100)), ('random', rng.normal(size=100))):
    for _ in range(2000):
        w = w - lr * H_small.T @ (H_small @ w - y_small)
    residual = np.linalg.norm(H_small @ w - y_small)
    print(f'start {name:6s}: residual {residual:.0e}, norm '
          f'{np.linalg.norm(w):5.2f}, distance to w* '
          f'{np.linalg.norm(w - w_star):.1f}')
```

```
start zero  : residual 3e-18, norm  0.45, distance to w* 0.0
start random: residual 2e-15, norm  8.45, distance to w* 8.4
```

Both runs fit every label. The run from zero ends at the minimum-norm solution of `lstsq` up to rounding; the run from a random start ends at another exact fit, more than ten times

longer. The minimum-norm solutions of the double-descent experiment are therefore the ones gradient descent from zero would find.

4.5.3.2 Maximum-Margin Classification

For classification with cross-entropy the situation differs. Take two classes with labels $y \in \{-1, +1\}$ and a linear score $\mathbf{w}^\top \mathbf{x}$, so that an example has the margin $m = y\mathbf{w}^\top \mathbf{x}$ and the loss $\log(1 + \exp(-m))$ of Section 3.1. If some $\mathbf{w}$ classifies every training example correctly, no finite $\mathbf{w}$ minimizes the loss: scaling up a separating $\mathbf{w}$ raises every margin and lowers every loss. Gradient descent lets $\|\mathbf{w}\|$ grow without bound, and the question becomes which direction $\mathbf{w}/\|\mathbf{w}\|$ it approaches. Soudry et al. (2018) proved that it approaches the *maximum-margin* direction: of all separators, the one whose smallest distance to a training point is largest (an example lies at distance $m/\|\mathbf{w}\|$ from the separator). The intuition is that once $\|\mathbf{w}\|$ is large, the loss of an example is about $\exp(-m)$, so the examples with the smallest margins dominate the gradient, and gradient descent works mostly on raising them. It is in no hurry: in Figure 4.11 the training error is zero from the first step, yet the separator is still turning after 10^5 steps.

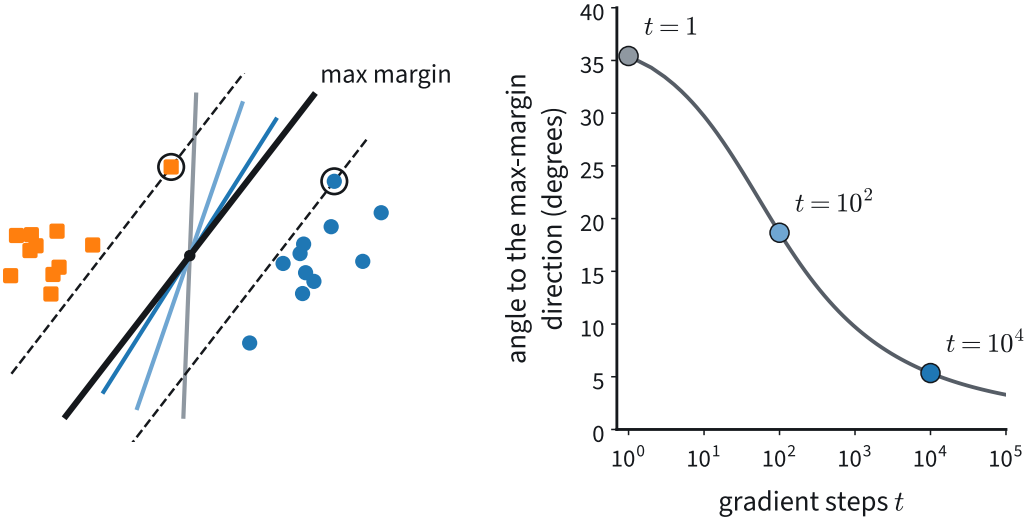


Figure 4.11: Gradient descent on the logistic loss for two separable classes, with separators through the origin. Left: the separators after $t = 1$, 10^2, and 10^4 steps (gray to blue) turn toward the maximum-margin separator (black), whose margin is dashed; the two examples closest to it are ringed. Right: the angle between $\mathbf{w}_t$ and the maximum-margin direction keeps shrinking long after every training example is classified correctly.

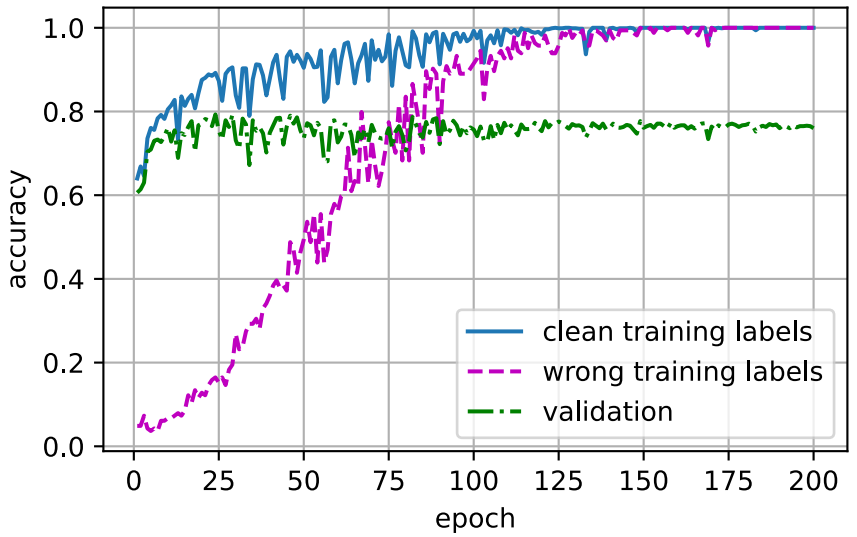
Both results concern linear models, and they prove nothing about our MLP. For deep networks, statements this exact are rare. The parameter norm that gradient descent keeps small is not even a property of the function the network computes, because different parameters can compute the same function (exercise 6). What we can do with the MLP is watch it.

4.5.4 Early Stopping

4.5.4.1 Label Noise

Gradient descent also has a preference in time. Arpit et al. (2017) observed that networks trained on partly mislabeled data fit the correctly labeled examples first and the mislabeled ones later. We repeat their experiment on our 1,000 images, redrawing 20% of the labels at random. A redrawn label is correct one time in ten, so about 18% of the labels end up wrong. After every epoch we measure the accuracy on the correctly labeled training images, on the mislabeled training images against their wrong labels, and on the 1,000 held-out training images, our validation set.

```
labels = corrupt(y_train, 0.2, rng)
wrong = labels != y_train
trainer = Tracker(max_epochs=200, datasets={
    'clean': ArrayData(X_train[~wrong], labels[~wrong], 1000),
    'mislabeled': ArrayData(X_train[wrong], labels[wrong], 1000),
    'validation': ArrayData(X_val, y_val, 1000),
    'test': ArrayData(X_test, y_test, 1000)})
trainer.fit(MLP(num_hiddens=256, lr=0.1), ArrayData(X_train, labels))
acc = trainer.accuracy
d2l.plot(list(range(1, 201)), [acc['clean'], acc['mislabeled'],
                               acc['validation']], 'epoch', 'accuracy',
         legend=['clean training labels', 'wrong training labels',
                 'validation'], figsize=(5, 3))
```



The two kinds of training images are fit at very different speeds. After 20 epochs the network reproduces the labels of more than 80% of the correctly labeled images, but those of fewer than one in five of the mislabeled ones. Validation accuracy peaks in this first phase, within the first 50 epochs. Then the network memorizes the wrong labels, nearly all of them after about 100 to 160 epochs, and validation accuracy declines by several points. Memorizing the mislabeled examples costs accuracy on new images.

4.5.4.2 The Stopping Rule

Early stopping exploits this order. It monitors the validation accuracy after every epoch and keeps the parameters of the best epoch. In practice, training ends once the validation accuracy has not improved for a number of epochs called the *patience* (exercise 2). Here we pick the best epoch from the recorded history and judge the choice on the test images, which played no part in it.

```
best = int(np.argmax(acc['validation']))
print(f'best validation accuracy after epoch {best + 1}: '
      f'test accuracy {acc["test"][best]:.3f}')
print(f'after all 200 epochs: test accuracy {acc["test"][-1]:.3f}')
```

```
best validation accuracy after epoch 24: test accuracy 0.778
after all 200 epochs: test accuracy 0.749
```

Stopping at the best validation epoch gains test accuracy over training to the end: several points as a rule, and anywhere between one and nine in our runs. And it takes less work, not more, since the remaining epochs need not even be computed. Early stopping limits how far the parameters move from their initialization, much as weight decay limits their size; for least squares the two are nearly equivalent (exercise 5).

The rule assumes that validation accuracy does not rise again after a long plateau. On small algorithmic tasks such as addition modulo a prime it does (Power et al. 2022). Networks trained with weight decay fit their training set within about a thousand steps and generalize only after a hundred to a thousand times as many (Figure 4.12), a phenomenon called *grokking*. A patience rule would have stopped long before.

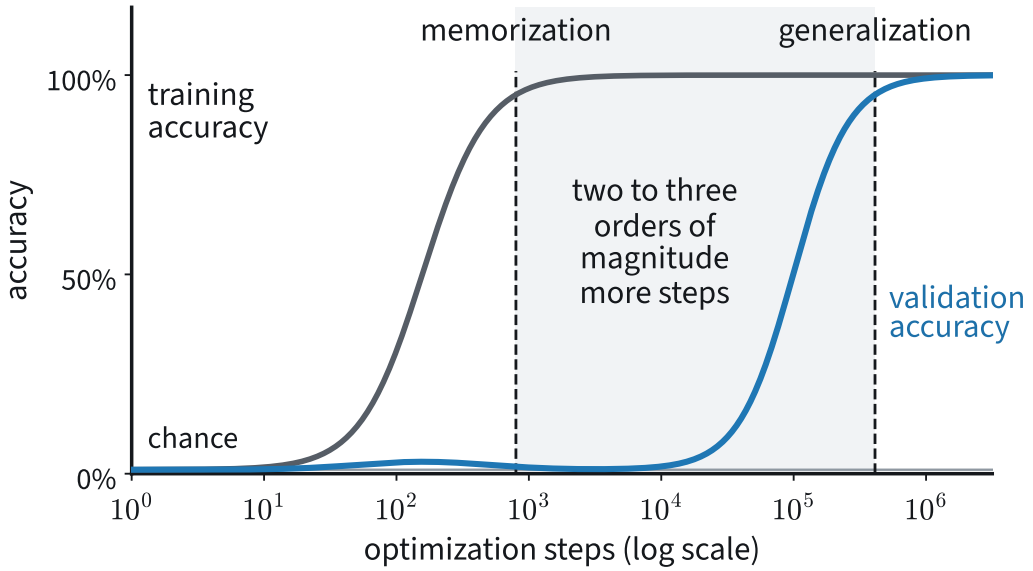


Figure 4.12: Grokking, schematically, after Power et al. (2022) . On a small algorithmic task, training accuracy (gray) saturates within about a thousand steps. Validation accuracy (blue) stays near chance for more than ten times as many steps and reaches the training accuracy two to three orders of magnitude after it.

How much of the opening question have we answered? For the two linear models we could say exactly which fit gradient descent finds. For the MLP we have seen one preference, clean examples before noisy ones, and no complete account of why its fits generalize exists yet. The last experiment does suggest where to push. If memorizing wrong labels is what costs accuracy, then a training procedure that makes memorizing harder should help, and the next section studies one: dropout.

4.5.5 Summary

An MLP with more parameters than training examples generalizes, although it can also fit random labels. No capacity measure of the class alone can therefore explain its generalization, and the VC bound is vacuous for it. A random-features model shows what happens past the interpolation threshold: test error peaks where the number of hidden units equals the number of examples, because the feature matrix is nearly singular there, and falls again as the width grows. Which of the many exact fits a model finds is decided by the training algorithm. For linear models we know how: gradient descent from zero finds the minimum-norm fit for least squares and approaches the maximum-margin separator for the logistic loss. For the MLP we know it only in part. With noisy labels it fits the correctly labeled examples first, so stopping at the best validation epoch ends training before most of the noise is memorized.

4.5.6 Exercises

1. **Limits of complexity measures.** Consider a generalization bound that holds for every data distribution: with probability at least $1 - \delta$ over the draw of n training examples, every classifier f in a class $\mathcal{F}$ satisfies

$$\epsilon(f) \leq \epsilon_{\mathcal{S}}(f) + B(\mathcal{F}, n, \delta). \tag{4.26}$$

   Here $\epsilon(f)$ is the population error, $\epsilon_{\mathcal{S}}(f)$ is the training error, and the term B depends only on $\mathcal{F}$, n, and δ, as the term $c\sqrt{(v + \log(1/\delta))/n}$ of the VC bound does. The text ran the random-label argument against such a bound for 1,000 images.

   1. Suppose that training also finds a member of $\mathcal{F}$ with zero training error on all 60,000 Fashion-MNIST training images when their labels are drawn uniformly at random, as Zhang et al. (2021) observed for image classifiers. What lower bound on $B(\mathcal{F}, 60{,}000, \delta)$ follows, and what does the bound then guarantee for the network trained on the true labels?
   2. Many later bounds let the second term depend on the classifier that training returns, for example through the norm of its weights or through its margins on the training examples. Which step of the argument no longer goes through? What must such a quantity do on the random-label run if the bound is to say something on the true labels?

2. code **Patience-based stopping.** `FashionMNIST` uses the Fashion-MNIST test set as its validation data, so hold out 10,000 training images as a validation set instead. Train an MLP with one hidden layer of 256 units, as in Section 4.2, on the remaining images for a fixed number of epochs large enough to overfit, and record the validation loss and the test accuracy after every epoch. A patience rule with tolerance θ and patience k stops once the validation loss has failed to improve on its best value by more than θ

for k consecutive epochs. It then retains the parameters of the epoch with the lowest validation loss so far.

1. Fix θ and apply the rule to the recorded curves for $k \in \{1, 5, 20\}$. For each k, report the stopping epoch, the retained epoch, and the test accuracy at both.
2. How does the test accuracy at the retained epoch depend on k? Relate the dependence to the epoch-to-epoch fluctuation of the validation loss.

Adapted from Michael Nielsen, Neural Networks and Deep Learning, Chapter 3.

3. code **More data can hurt.** Fix the random-features model at 1,000 hidden units and fit it, as in Section 4.5.2, on the first n of 2,000 training images for n from 100 to 2,000.

1. Plot the test error against n. Where does it peak, and how does the peak compare with the error at $n = 2{,}000$?
2. Plot the smallest singular value of the feature matrix against n and explain the shape of the error curve with it.
3. Repeat with 500 and 2,000 hidden units. Where does the peak move?

4. code **Weight decay at the threshold.** Replace the minimum-norm fit of the double-descent experiment by the ridge solution $\mathbf{W}^{(2)} = (\frac{1}{n}\mathbf{H}^\top\mathbf{H} + \lambda\mathbf{I})^{-1}\frac{1}{n}\mathbf{H}^\top\mathbf{Y}$ of Section 2.7, which, as there, averages the squared error over the n examples.

1. Plot the test error against the width for $\lambda \in \{10^{-5}, 10^{-3}, 10^{-1}\}$. What happens to the peak at $h = n$?
2. Show that the ridge solution approaches the minimum-norm solution as $\lambda \to 0$ for $h > n$. Hint: write $(\frac{1}{n}\mathbf{H}^\top\mathbf{H} + \lambda\mathbf{I})^{-1}\frac{1}{n}\mathbf{H}^\top = \mathbf{H}^\top(\mathbf{H}\mathbf{H}^\top + n\lambda\mathbf{I})^{-1}$.
3. For each width, choose λ on the 1,000 validation images. Does the curve of the tuned model still descend twice?

5. **Early stopping and weight decay.** ▯ Early stopping limits the number of training epochs where weight decay limits the weights. Linear least squares makes this precise. Let $\mathbf{X}^\top\mathbf{X}/n$ have eigenvalues $\mu_i > 0$, let $\mathbf{w}^*$ minimize $L(\mathbf{w}) = \frac{1}{2n}\|\mathbf{X}\mathbf{w} - \mathbf{y}\|^2$, and run gradient descent on L from $\mathbf{w}_0 = \mathbf{0}$ with learning rate $\eta < 1/\max_i \mu_i$.

1. Show that after t steps the coordinate of $\mathbf{w}_t$ along the i-th eigenvector equals $1 - (1 - \eta\mu_i)^t$ times that of $\mathbf{w}^*$.
2. The minimizer of $L(\mathbf{w}) + \frac{\lambda}{2}\|\mathbf{w}\|^2$ scales the same coordinate by $\mu_i/(\mu_i + \lambda)$ (Section 2.7.3). For which λ, as a function of η and t, do the two factors approximately agree both in the directions with $\eta\mu_i t \ll 1$ and in those with $\eta\mu_i t \gg 1$?

Adapted from Ian Goodfellow, Yoshua Bengio, and Aaron Courville, Deep Learning, Section 7.8.

6. **Implicit bias of the logistic loss.** ▯ For linearly separable data, gradient descent on the logistic loss converges in direction to the maximum-margin separator (Soudry et al. 2018). Take labels $y_i \in \{-1, +1\}$.

1. For a linear predictor $f(\mathbf{x}) = \mathbf{w}^\top\mathbf{x}$, show that on separable data the loss $\sum_i \log\left(1 + \exp(-y_i\mathbf{w}^\top\mathbf{x}_i)\right)$ is positive for every $\mathbf{w}$ but has infimum zero. Conclude that $\|\mathbf{w}_t\|$ grows without bound when gradient descent drives the loss to zero, so that only the direction $\mathbf{w}_t/\|\mathbf{w}_t\|$ can converge.
2. For a two-layer ReLU network $f(\mathbf{x}) = \sum_j v_j \max(0, \mathbf{w}_j^\top\mathbf{x})$, show that replacing $(\mathbf{w}_j, v_j)$ by $(c_j\mathbf{w}_j, v_j/c_j)$ with $c_j > 0$ leaves f unchanged. Find the smallest value

of $\sum_j (\|\mathbf{w}_j\|^2 + v_j^2)$ over these rescalings. Explain why a margin measured with the parameter norm of the network, unlike one measured with $\|\mathbf{w}\|$ for the linear predictor, is not a property of the function f alone.

7. [extended] **Epoch-wise double descent.** ▯ Take an MLP with one hidden layer of 256 units, as in Section 4.2, on Fashion-MNIST and randomly relabel 15% of the training examples. Train far past convergence (several hundred epochs). After every epoch, record the test error and the training accuracy on the relabeled examples, measured against their new labels. Plot both against the epoch count on a log axis.
   1. Does the test error descend a second time after the initial rise?
   2. At which epoch does the test error peak, and what fraction of the relabeled examples has the network fitted by then?
   3. Repeat with 5% and 30% of the examples relabeled. How do the height and the epoch of the peak change?
   4. Suppose a patience rule with $k = 20$ and $\theta = 0$ monitored the test-error curve of the 15% run as if it were a validation curve. At which epoch would the rule stop, and how does the test error there compare with the test error at the end of training?

8. [extended] **Grokking.** ▯▯ Reproduce the setup of Power et al. (2022): train a small network (they use a two-layer transformer, but a wide MLP on one-hot pairs also works) to predict $c = (a + b) \bmod 97$ from the pair (a, b), using a random 50% of all pairs for training, with weight decay, for 10^5 or more steps. Plot training and validation accuracy against the logarithm of the step count, and compare with Figure 4.12. How does the delay before generalization change with the training fraction and with the weight-decay strength?

4.6 Dropout

An MLP left to train long enough memorizes whatever labels it is given, wrong ones included, and it loses accuracy on new images as it does so (Section 4.5). The two regularizers we have met so far stand in its way differently: weight decay (Section 2.7) penalizes large weights, and early stopping (Section 4.5.4) ends training before the mislabeled examples are memorized. *Dropout* takes a third route, and a rather drastic one. It disturbs the network while it learns. At every training step, each hidden unit is set to zero with some probability, independently of the others, and the network computes its output from the units that remain. At test time the full network runs.

The idea came from a suspicion about how networks overfit. Neither weight decay nor early stopping has anything to say about how the hidden units divide the work among themselves. A unit may become useful only together with particular other units, for instance because it corrects their errors on the training images, and there is no reason why such arrangements should carry over to new images. Hinton et al. (2012) called this *co-adaptation*. If units come and go at random, no unit can count on any other being present.

We state the operation, implement it in an MLP with two hidden layers, and train that MLP on Fashion-MNIST. To see what dropout changes, we then return to the setting in which the last section watched memorization happen, a small training set with many wrong labels. Finally we ask why dropout has the effect it has. Three explanations are on offer, and we shall have to be careful with all of them.

4.6.1 Noise in Hidden Layers

4.6.1.1 Simplicity and Robustness

So far we have measured the simplicity of a model in two ways: by the number of its features (Section 2.6.2.1) and by the norm of its weights, which weight decay keeps small. A third measure is robustness. A good image classifier should not change its prediction when a little noise is added to the pixels, so we may train it on noisy pixels and ask it to predict the clean label.

For a linear model this turns out to be nothing new. Add to the input a noise vector ϵ whose entries are independent with mean zero and variance s^2, and the expected squared error grows by $s^2\|\mathbf{w}\|^2$, the variance of $\mathbf{w}^\top\epsilon$: training on noisy inputs is weight decay. For nonlinear models and small noise, the penalty is approximately one on how fast the output changes with the input (Bishop 1995). Dropout carries the idea from the inputs to the hidden units, and it changes the noise: instead of nudging a unit, it removes it. Noise of that size lies outside Bishop's analysis, so we take from it the motivation and no more.

4.6.1.2 Masking and Rescaling

Removing units shrinks the input of the next layer. With half of the units set to zero, a weighted sum over them is on average half as large, so dropout scales up the units that it keeps. Consider one hidden unit, whose activation h_j is entry j of the hidden representation $\mathbf{h}$ of an example. At each training step, dropout draws a random *mask* ξ_j, equal to 0 with probability p and to 1 with probability $1-p$, independently for every unit and every example, and replaces h_j by

$$h'_j = \frac{\xi_j h_j}{1-p} = \begin{cases} 0 & \text{with probability } p, \\ \dfrac{h_j}{1-p} & \text{with probability } 1-p. \end{cases} \tag{4.27}$$

The number p is the *dropout probability*. Does the factor $1/(1-p)$ make up for the dropped units? For a fixed h_j, the expected value over the mask is

$$\mathbb{E}[h'_j] = p \cdot 0 + (1-p)\frac{h_j}{1-p} = h_j,$$

so on average the next layer receives the activation it would receive without dropout, and $1/(1-p)$ is the only factor with this property. The fluctuations around this average are large. Since $\mathbb{E}[h'^2_j] = (1-p)\,h_j^2/(1-p)^2 = h_j^2/(1-p)$, the variance is

$$\mathrm{Var}[h'_j] = \mathbb{E}[h'^2_j] - h_j^2 = \frac{p}{1-p}\,h_j^2. \tag{4.28}$$

The noise grows with p and with the size of the activation. For $p = 0.5$ a kept unit is doubled and a dropped one vanishes, so the standard deviation of h'_j equals $|h_j|$. Unlike the small input noise above, dropout perturbs a unit by as much as its own value.

The original formulation (Hinton et al. 2012) left the kept activations unscaled and instead multiplied the outgoing weights of each unit by $1-p$ at test time. Deep learning frameworks

rescale during training, as in Equation 4.27, a version called *inverted dropout*. Its merit is practical: the network at test time is the plain network, with no dropout code at all.

4.6.1.3 Training and Inference

At test time we want a deterministic prediction, so dropout is switched off: the full network runs, with no masks and no rescaling. Does it compute the average of the random networks seen in training? For a single unit it does, since each hidden activation equals the expected value of its dropped version. This remains true through the next linear layer: a pre-activation $z = \sum_j w_j h'_j$ has $\mathbb{E}[z] = \sum_j w_j h_j$, the value of the full network. The activation function that follows is not linear, however, and the expected value does not pass through it.

A small example shows the difference. Let two units with activations $h_1 = h_2 = 1$ feed the pre-activation $z = h'_1 - h'_2$, with $p = 0.5$. Each h'_j is 0 or 2 with equal probability, so z takes the values -2, 0, 0, 2, each with probability $1/4$, and $\mathbb{E}[z] = 0$, the value of the full network. After a ReLU, however, the average over the masks is $\mathbb{E}[\mathrm{ReLU}(z)] = \frac{1}{4} \cdot 2 = 0.5$, whereas the full network computes $\mathrm{ReLU}(0) = 0$. In a network with nonlinear layers, the network at test time therefore does not compute the average output of the networks that dropout samples during training. It approximates that average, and we return to the quality of the approximation at the end of the section. Averaging the outputs of many passes with dropout switched on estimates the average itself (exercise 5).

4.6.2 Implementation

4.6.2.1 The Dropout Layer

To drop the entries of a tensor `X` with probability `dropout`, we draw a number uniformly from $[0, 1)$ for each entry and keep the entry if the number is at least `dropout`, which happens with probability $1 -$ `dropout`. Multiplying `X` by this 0/1 mask and dividing by $1 -$ `dropout` applies Equation 4.27 to every entry at once. The case `dropout` $= 1$ drops everything and is handled separately, to avoid a division by zero.

```
def dropout_layer(X, dropout):
    assert 0 <= dropout <= 1
    if dropout == 1: return torch.zeros_like(X)
    mask = torch.rand_like(X) >= dropout
    return mask.to(X.dtype) * X / (1.0 - dropout)
```

We apply `dropout_layer` to a 2×8 tensor with the probabilities 0, 0.5, and 1. Then we drop the entries of a vector of a million ones with probability 0.2 and compare the mean and the variance of the result with the values 1 and $p/(1-p) = 0.25$ computed above.

```
X = d2l.reshape(d2l.arange(16, dtype=d2l.float32), (2, 8))
for p in (0, 0.5, 1):
    print(f'p = {p}:', dropout_layer(X, p))
Y = dropout_layer(d2l.ones(10**6), 0.2)
print(f'mean {float(d2l.reduce_mean(Y)):.3f}, '
      f'variance {float(d2l.reduce_mean((Y - 1) ** 2)):.3f}')
```

```
p = 0: tensor([[ 0.,  1.,  2.,  3.,  4.,  5.,  6.,  7.],
        [ 8.,  9., 10., 11., 12., 13., 14., 15.]])
p = 0.5: tensor([[ 0.,  2.,  4.,  6.,  0.,  0.,  0.,  0.],
        [16.,  0., 20.,  0.,  0., 26., 28., 30.]])
p = 1: tensor([[0., 0., 0., 0., 0., 0., 0., 0.],
        [0., 0., 0., 0., 0., 0., 0., 0.]])
mean 1.000, variance 0.250
```

With $p = 0.5$, each entry is either zeroed or doubled, and how many of the 16 are zeroed varies from run to run. The million entries dropped with $p = 0.2$ have mean 1 and variance 0.25 up to about 0.001, as computed.

4.6.2.2 The Model

We apply dropout to the output of each hidden layer, after the activation function. Figure 4.13 shows the effect on the MLP of Figure 4.1, with four inputs, five hidden units, and three outputs, for one draw of the masks, in which h_2 and h_5 are dropped. The outputs no longer depend on these two units. Backpropagation sends zero gradient through a multiplication by zero, so the weights into and out of h_2 and h_5 receive no update from this example. The next step draws new masks and removes other units.

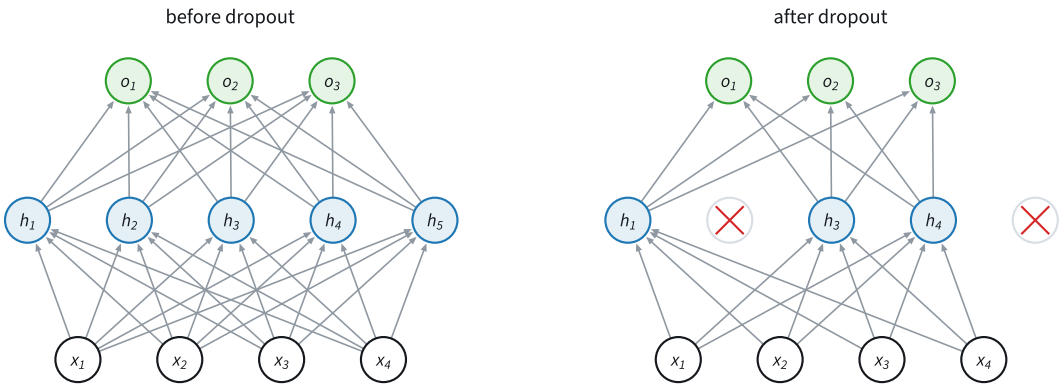


Figure 4.13: Dropping the hidden units h_2 and h_5 of an MLP removes them together with their connections.

Our model has two hidden layers of 256 ReLU units each. Each hidden layer may have its own dropout probability. A common choice drops fewer units in the layers close to the input; we use $p = 0.2$ after the first hidden layer and $p = 0.5$ after the second. We register the parameters with the framework as in Section 3.3 and give them the He initialization of Section 4.4.2.2: normal weights with variance $2/n_{\text{in}}$, where n_{in} is the number of inputs of the layer, and biases of 0. The model applies dropout only while it is being trained.

`self.training` is set by `train()` and `eval()`, which the `Trainer` calls before updating and before evaluating the model.

```
class DropoutMLPScratch(d2l.Classifier):
    def __init__(self, num_outputs, num_hiddens_1, num_hiddens_2,
                 dropout_1, dropout_2, lr, num_inputs=784):
        super().__init__()
```

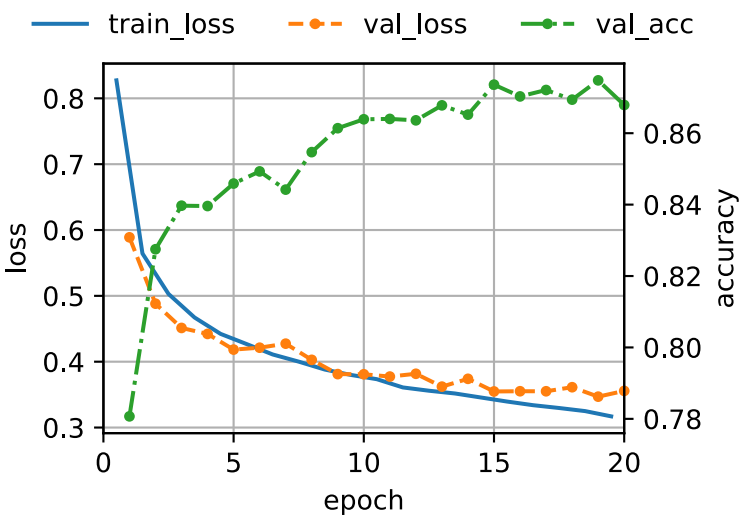
```
        self.save_hyperparameters()
        he = lambda n_in, n_out: nn.Parameter(  # Variance 2 / n_in
            d2l.normal(0, (2 / n_in) ** 0.5, (n_in, n_out)))
        self.W1 = he(num_inputs, num_hiddens_1)
        self.W2 = he(num_hiddens_1, num_hiddens_2)
        self.W3 = he(num_hiddens_2, num_outputs)
        self.b1 = nn.Parameter(d2l.zeros(num_hiddens_1))
        self.b2 = nn.Parameter(d2l.zeros(num_hiddens_2))
        self.b3 = nn.Parameter(d2l.zeros(num_outputs))

    def forward(self, X):
        X = X.reshape(-1, self.num_inputs)
        H1 = torch.relu(X @ self.W1 + self.b1)
        if self.training:
            H1 = dropout_layer(H1, self.dropout_1)
        H2 = torch.relu(H1 @ self.W2 + self.b2)
        if self.training:
            H2 = dropout_layer(H2, self.dropout_2)
        return H2 @ self.W3 + self.b3
```

4.6.2.3 Training

We train the model like softmax regression in Section 3.4, with minibatch SGD at learning rate 0.1 and minibatches of 256 images, here for 20 epochs.

```
hparams = {'num_outputs': 10, 'num_hiddens_1': 256, 'num_hiddens_2': 256,
           'dropout_1': 0.2, 'dropout_2': 0.5, 'lr': 0.1}
model = DropoutMLPScratch(**hparams)
data = d2l.FashionMNIST(batch_size=256)
trainer = d2l.Trainer(max_epochs=20, num_gpus=1)
trainer.fit(model, data)
```



After 20 epochs the validation accuracy is about 87%, and it moves by a point or more from one epoch to the next, as it did in Section 4.2. The training loss needs a word of explanation. It is computed with the masks switched on, so it measures the thinned networks, which predict worse than the full network that the validation loss measures. For the first few epochs the training loss therefore lies above the validation loss, which looks like a bug and is not. It ends a little below it.

4.6.2.4 Concise Implementation

The framework provides dropout as a layer that takes the dropout probability as its argument. The layer applies Equation 4.27 while the model is being trained and passes its input through unchanged otherwise, so the model needs no test for the training mode. We place a dropout layer after the activation function of each hidden layer and ask the framework for the initialization of the scratch model, normal weights with variance $2/n_{\text{in}}$.

The `Trainer` initializes the lazy layers through the model's `weight_init` method, as in Section 3.4.

```
class DropoutMLP(d2l.Classifier):
    def __init__(self, num_outputs, num_hiddens_1, num_hiddens_2,
                 dropout_1, dropout_2, lr):
        super().__init__()
        self.save_hyperparameters()
        self.net = nn.Sequential(
            nn.Flatten(), nn.LazyLinear(num_hiddens_1), nn.ReLU(),
            nn.Dropout(dropout_1), nn.LazyLinear(num_hiddens_2), nn.ReLU(),
            nn.Dropout(dropout_2), nn.LazyLinear(num_outputs))

    def weight_init(self, module):
        if isinstance(module, nn.Linear):
            nn.init.kaiming_normal_(module.weight, nonlinearity='relu')
            nn.init.zeros_(module.bias)
```

We do not train this model here. The experiment that follows does.

4.6.3 Effects of Dropout

4.6.3.1 Too Little to Regularize

Does dropout make our MLP more accurate? On all 60,000 training images the question has little to bite on. A regularizer can only remove overfitting that is there, and after 20 epochs the training loss of our network, even with the masks switched on, ends only a little below the validation loss. To see what dropout does, we need a network that overfits. Section 4.5.4 gave us one: an MLP trained for 200 epochs on 1,000 images, part of whose labels had been redrawn at random, memorized the wrong labels and lost validation accuracy doing so.

4.6.3.2 Noisy Labels

We repeat that experiment with the MLP of this section and make it harsher: we redraw 40% of the labels, so that about 36% of them are wrong. More than one training image in three now teaches the network something false. The helpers of Section 4.5.4 do the work: `d2l.corrupt` redraws the labels, `d2l.ArrayData` serves arrays as minibatches, and `d2l.Tracker`

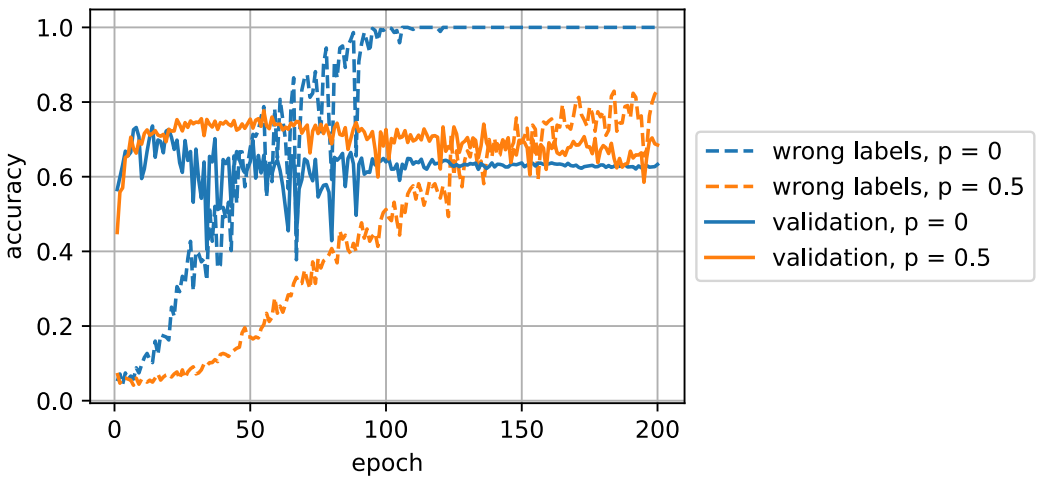
records the accuracy on named datasets after every epoch. We validate on the first 2,000 images of the validation set.

```
images = d2l.FashionMNIST(batch_size=2000)
X, y = (d2l.numpy(a) for a in next(iter(images.train_dataloader())))
X_val, y_val = (d2l.numpy(a) for a in next(iter(images.val_dataloader())))
X, y, X_val = X[:1000].reshape(1000, -1), y[:1000], X_val.reshape(2000, -1)
labels = d2l.corrupt(y, 0.4, np.random.default_rng())
wrong = labels != y  # The training labels that are now wrong
```

We train the concise model twice for 200 epochs, with minibatches of 64, once with dropout probability 0.5 in both hidden layers and once without dropout. After every epoch we record how many of the wrong labels the network reproduces and its accuracy on the validation images.

```
acc = {}
for p in (0, 0.5):
    trainer = d2l.Tracker(max_epochs=200, datasets={
        'wrong labels': d2l.ArrayData(X[wrong], labels[wrong], 1000),
        'validation': d2l.ArrayData(X_val, y_val, 1000)})
    trainer.fit(DropoutMLP(**{**hparams, 'dropout_1': p, 'dropout_2': p}),
                d2l.ArrayData(X, labels))
    acc[p] = trainer.accuracy
    print(f'dropout {p}: wrong labels reproduced '
          f'{acc[p]["wrong labels"][-1]:.2f}, validation accuracy over the '
          f'last 10 epochs {np.mean(acc[p]["validation"][-10:]):.3f}')
d2l.plot(list(range(1, 201)), [acc[0]['wrong labels'], acc[0.5]['wrong labels'],
                               acc[0]['validation'], acc[0.5]['validation']],
         'epoch', 'accuracy', fmts=('C0--', 'C1--', 'C0-', 'C1-'),
         legend=['wrong labels, p = 0', 'wrong labels, p = 0.5',
                 'validation, p = 0', 'validation, p = 0.5'],
         legend_loc='outside', figsize=(4.5, 3))
```

```
dropout 0: wrong labels reproduced 1.00, validation accuracy over the last 10 epochs 0.627
dropout 0.5: wrong labels reproduced 0.83, validation accuracy over the last 10 epochs 0.664
```



Without dropout, the network reproduces every wrong label by about epoch 100, and its validation accuracy falls as it memorizes them, by about ten points from its early best. With dropout, the wrong labels are learned far more slowly: after 200 epochs the network reproduces only three quarters to four fifths of them, and over the last ten epochs its validation accuracy is three to five points higher than without dropout. Here, then, dropout does what a regularizer should. It changes which fit training finds when there is noise to fit.

Note also what dropout has not done. It slows the memorization and does not stop it. The curve of the wrong labels is still rising after 200 epochs, and the network trained with dropout loses validation accuracy too, only later and less. Neither network ends as accurate as it was within its first fifty epochs, so dropout has not made early stopping superfluous.

4.6.4 Interpretations

Why does dropout help, here and on the benchmarks of images, speech, and text on which Srivastava et al. (2014) reported lower test errors with it? Our experiment shows what dropout does, not why. Three accounts are standard, and we present them as accounts: each describes part of the mechanism, and none is complete.

4.6.4.1 Noise Injection

The first account takes dropout as noise: a network that must tolerate noise in its hidden units is pushed toward functions that are insensitive to it. For a linear model we can say what this means. Apply dropout to the inputs of the linear model $\mathbf{w}^\top \mathbf{x} + b$ and train it with the squared loss. The dropped inputs x'_j are independent, each has mean x_j and variance $\frac{p}{1-p} x_j^2$, so the prediction has mean $\mathbf{w}^\top \mathbf{x} + b$ and variance $\frac{p}{1-p} \sum_j w_j^2 x_j^2$. The expected loss is the squared error of the mean plus this variance,

$$\mathbb{E}\left[\left(\mathbf{w}^\top \mathbf{x}' + b - y\right)^2\right] = \left(\mathbf{w}^\top \mathbf{x} + b - y\right)^2 + \frac{p}{1-p} \sum_j x_j^2 w_j^2. \tag{4.29}$$

In expectation, dropout on the inputs of a linear model is thus a weight decay (Srivastava et al. 2014), though not the plain one that additive noise gave us. Summed over the training examples, the penalty on w_j is proportional to the sum of the squares of feature j: weights of features that are often large are penalized more, and the weight of a feature that is always 0 is not penalized at all. This is as far as the calculation goes. In a deep network the penalty depends on the hidden activations, which change during training, and no such closed form exists.

4.6.4.2 Co-adaptation

The second account is the one for which dropout was invented, and with which this section began. Units that cannot count on their partners must each be useful on their own, and such units should carry over to new data better than fragile arrangements of many. The account is plausible. Our experiment neither supports nor refutes it: any regularizer that slows the memorization of wrong labels would have shown the same curves. We have used the idea as a motivation, and it should not be taken for more than that.

4.6.4.3 An Implicit Ensemble

Our network has 512 hidden units and hence 2^{512} masks, each of which defines a *thinned* network, and all of them share one set of weights. Every training step trains as many of them as the minibatch has examples, each on one example. The network at test time can be read as an approximation of the average of their predictions. Averaging the predictions of many models, an *ensemble*, typically predicts better than any single one of them, so dropout would obtain such an average while training a single network. The reading is loose. The thinned networks are not trained independently of each other, as the members of an ensemble are, and the two-unit example above showed that the full network does not compute their average. When the only dropped units feed the output layer directly, the full network computes a weighted geometric mean of the thinned networks' predictions (exercise 7); deeper in the network, not even that holds.

4.6.4.4 Dropout in Practice

Where, then, should we expect dropout to matter? Where our experiment found it to: many more parameters than examples, and many passes over each example. This was the situation of the large fully connected layers in the image classifiers of the early 2010s, among them AlexNet (Section 7.1), for which dropout was introduced. Later architectures use it at small probabilities, around 0.1, and models that see each training example only once, as large language models do, are often trained without it. One caution: during training the mean square of a dropped unit is $1/(1-p)$ times that of the unit itself, and layers that normalize their inputs (Section 7.3) are sensitive to this mismatch between training and test (Li et al. 2019).

The next section meets many parameters and few examples outside images: an MLP with nineteen parameters for each of the houses whose prices it is to predict.

4.6.5 Summary

Dropout sets each hidden unit to zero with probability p at every training step and scales the remaining units by $1/(1-p)$, so that each unit keeps its expected value; at test time the full network runs. The expected value is preserved only unit by unit, so the full network is not the average of the networks that dropout trains. On all of Fashion-MNIST our MLP overfits too little for dropout to matter. Trained on 1,000 images with many wrong labels, the MLP without dropout memorizes all of them and loses validation accuracy; with dropout it memorizes them more slowly and keeps more of its accuracy. For a linear model, dropout of the inputs is, in expectation, a weight decay that penalizes each weight by the square of its input. For deep networks, noise, co-adaptation, and the implicit ensemble are partial accounts of its effect.

4.6.6 Exercises

1. code **Dropout probabilities.** The model of this section drops units with probability 0.2 after the first hidden layer and 0.5 after the second.
    1. Train it with the two probabilities swapped, with both set to 0.5, and with both set to 0.8. For each run, report the best validation accuracy. Which layer tolerates the higher probability?
    2. The `Trainer` computes the training loss that it plots with the masks switched on. After training, evaluate the loss on the training images with the masks switched

off, and compare the gap between training and validation loss for the networks trained with and without dropout.

2. **Dropout and He initialization.** The He initialization of Section 4.4.2.2 chooses the variance of the weights so that the mean square of the pre-activations stays the same from layer to layer.

    1. During training, by what factor does dropout with probability p change the mean square of the pre-activations of the next layer? Give the factor for each of the two dropout layers of this section's model.
    2. In a network with $K - 1$ hidden layers, each followed by dropout with probability p, how does the factor grow with K? Should the initialization correct for it, given that the network runs without dropout at test time? Relate your answer to the mismatch between training and test that dropout causes before a batch normalization layer.

3. code **Less noise, less dropout.** The noisy-label experiment of this section redrew 40% of the labels and dropped units with probability 0.5 in both hidden layers.

    1. Repeat it with 20% of the labels redrawn, as in Section 4.5.4, once with probability 0.5 in both layers and once with the 0.2 and 0.5 of the full-data run. In each case, does dropout still raise the validation accuracy of the last epochs, or does it only delay the memorization?
    2. In every run, with and without dropout, find the best validation accuracy and the epoch at which it is reached. Which helps more against wrong labels, dropout or stopping early? Do the two combine?

4. **Ensemble size.** The two hidden layers of this section's model have 512 units together, so they admit 2^{512} joint dropout masks, and the ensemble interpretation views each mask as a thinned network.

    1. Training runs for 20 epochs over the 60,000 Fashion-MNIST training images, and a fresh mask is drawn for every example. How many joint masks are sampled during training, and what fraction of all joint masks is that? How likely is it that any mask is drawn twice? The masks are not equally likely, since the first layer drops units with probability 0.2: bound the probability that two draws coincide by the probability of the most likely joint mask.
    2. Using these numbers, explain in what sense dropout does, and does not, train an ensemble of all these networks.

5. code **Monte Carlo dropout.** Instead of switching dropout off at test time, keep it on in the trained `DropoutMLPScratch` and run $T = 20$ forward passes per validation image, then average the softmax outputs (Gal and Ghahramani 2016). Compare the accuracy and the calibration (the mean of the largest softmax probability against the accuracy) with those of the single pass of the full network. How does the accuracy change as T grows from 1 to 20?

6. code **Dropout and weight decay.** Weight decay and dropout both regularize the model of this section, by different mechanisms. Measure each run by its best validation accuracy and by the gap between training and validation loss, with both losses evaluated with the masks switched off.

    1. With both dropout probabilities set to zero, add weight decay as in Section 2.7 at strengths $\lambda \in \{10^{-4}, 10^{-3}, 10^{-2}\}$. Compare these runs with the runs with and

without dropout of this section.
2. Keep the dropout probabilities at 0.2 and 0.5, and add weight decay at the same strengths. Are the effects of the two regularizers additive, do they show diminishing returns, or do they cancel each other out?

7. **Dropout at test time.** ☒ Set `dropout_1` to zero, so that only the second hidden layer is masked, and let ξ range over its masks. Let $\mathbf{o}(\xi)$ and $\hat{\mathbf{y}}(\xi)$ be the logits and the softmax output under mask ξ, which occurs with probability $P(\xi)$. Show that the logits of the full network equal $\sum_\xi P(\xi)\,\mathbf{o}(\xi)$. Show further that its softmax output equals the normalized weighted geometric mean

$$\tilde{y}_j = \frac{\prod_\xi \hat{y}_j(\xi)^{P(\xi)}}{\sum_k \prod_\xi \hat{y}_k(\xi)^{P(\xi)}}. \tag{4.30}$$

Explain why neither equality holds in general once `dropout_1` is nonzero.

4.7 Predicting House Prices on Kaggle

Fashion-MNIST arrives clean: every example is an image of the same size, every pixel lies between 0 and 1, and every label comes from the same ten classes. Most data that people want to predict from looks different. In this section we predict the sale prices of houses in Ames, Iowa, from a table in which each house is described by 79 attributes. Some are numbers, such as the living area; others are categories, such as the type of roof; many entries are missing; and the numbers range from 0 or 1 to tens of thousands. Only 1,460 houses come with a price. A model cannot read such a table directly, and with so few examples, a careless estimate of its error can be off by more than the differences between the models we want to compare.

The task brings together much of what the book has covered so far: the squared loss and minibatch SGD of Chapter 2, cross-validation (Section 2.6) and weight decay (Section 2.7), and the MLP of this chapter with the initialization of Section 4.4. Most of the work lies in preparing the data and in estimating the error; the models take a few lines each.

```
import pandas as pd
```

4.7.1 The Competition

4.7.1.1 Kaggle

Kaggle hosts machine learning competitions. Each competition provides a training set with labels and a test set without them. Participants upload predictions for the test set, and Kaggle scores them against labels that it keeps hidden, so no participant can tune a model on the test labels. The scores of all participants appear on a leaderboard. The data of this section belong to the competition House Prices: Advanced Regression Techniques, a practice competition without a deadline. The data were collected by De Cock (2011) from the sales of 2006 to 2010, as a larger and more varied replacement for small teaching datasets such as Boston Housing.

4.7.1.2 The Data

We download the training and the test table with `d2l.download` and read them with `pandas` (Section 1.2).

```
def read_house_prices(root='../data'):
    """Download and read the training and test tables of the competition."""
    train = pd.read_csv(d2l.download(
        d2l.DATA_URL + 'kaggle_house_pred_train.csv', root,
        sha1_hash='585e9cc93e70b39160e7921475f9bcd7d31219ce'))
    test = pd.read_csv(d2l.download(
        d2l.DATA_URL + 'kaggle_house_pred_test.csv', root,
        sha1_hash='fa19780a7b011d9b009e8bff8e99922a8ee2eb90'))
    return train, test
```

The training table has one row for each of 1,460 houses: an identifier, 79 attributes, and the sale price, the label. The test table has 1,459 houses, with identifiers and attributes but no prices.

```
raw_train, raw_test = read_house_prices()
print(raw_train.shape, raw_test.shape)
```

```
(1460, 81) (1459, 80)
```

Here are the first four houses, with the first three attributes, the last two, and the price:

```
print(raw_train.iloc[:4, [0, 1, 2, 3, -3, -2, -1]])
```

```
   Id  MSSubClass MSZoning  LotFrontage SaleType SaleCondition  SalePrice
0   1          60       RL         65.0       WD        Normal     208500
1   2          20       RL         80.0       WD        Normal     181500
2   3          60       RL         68.0       WD        Normal     223500
3   4          70       RL         60.0       WD       Abnorml     140000
```

`MSSubClass` encodes the type of dwelling as a number, `MSZoning` the zoning of the lot as a category, and `LotFrontage` the length of the lot's street front in feet; `SaleType` and `SaleCondition` describe the sale. Of the 79 attributes, 36 are stored as numbers and 43 as categories, and 19 have missing entries. Most of these mean that a feature is absent: `PoolQC`, the quality of the pool, is missing for 1,453 houses, which simply have no pool. The first column, `Id`, numbers the rows. The competition defines it as an identifier, so we do not use it as a feature.

4.7.1.3 The Error Measure

Prices span a wide range: the cheapest house in the training table sold for $34,900, the most expensive for $755,000. An error of $50,000 is more than the first house is worth and less than 7% of the second, so what matters is the relative error, not the error in dollars. The competition therefore measures the error on a logarithmic scale. If a prediction $\hat{y}$ satisfies $|\log \hat{y} - \log y| \leq \delta$, then $e^{-\delta} \leq \hat{y}/y \leq e^{\delta}$, and for small δ the prediction lies within a fraction of about δ of the true price y. Over the n houses of the test set, the score is the root-mean-squared error (RMSE) of the logarithms,

$$\sqrt{\frac{1}{n}\sum_{i=1}^{n}\left(\log y_i - \log \hat{y}_i\right)^2}. \tag{4.31}$$

An RMSE of 0.1 thus corresponds to typical errors of about 10% of the price. We train our models to predict $\log y$ with the squared loss, so the training loss is the square of the score computed on the training houses.

4.7.2 Preprocessing

4.7.2.1 Numerical Features

The numerical attributes have very different scales: the number of fireplaces is a small integer, the area of the lot runs to tens of thousands of square feet. A weight that suits one of them is far too large or too small for another, which slows minibatch SGD. Weight decay adds a second reason. It penalizes all weights alike, and so it would fall hardest on the small-valued features, which need the largest coefficients. We therefore standardize each numerical feature as in Section 1.2: with μ and s the mean and the standard deviation of the feature over the training rows, we replace its value x by $(x - \mu)/s$.

A missing numerical value is then replaced by 0, the mean of the standardized feature. This is the mean imputation of Section 1.2, and it assumes that a missing value is typical. For `LotFrontage`, missing for 259 houses, this is a fair guess, although the variance of the filled column falls below 1, to about 0.82. For the year in which a garage was built it is a poor one, since there a gap means that the house has no garage. Exercise 1 examines the bias that mean imputation introduces in such a case.

4.7.2.2 Categorical Features

A category cannot be multiplied by a weight. We therefore one-hot encode each categorical feature with `get_dummies`, as in Section 1.2. `MSZoning` takes five values in the training table, among them `RL` (residential, low density) and `RM` (residential, medium density), so it becomes five indicator features `MSZoning_RL`, `MSZoning_RM`, and so on. A missing value counts as one more category, so that "no pool" gets its own column.

A column stored as numbers is not always a quantity. `MSSubClass` codes the type of dwelling with numbers from 20 to 190: 20 stands for a one-story house built in 1946 or later, 60 for a two-story house of the same period, and 90 for a duplex. The differences between these codes measure nothing, but with one weight on the standardized column, the prediction of a linear model would change in proportion to them. We therefore convert the codes to strings before the encoding, and `MSSubClass` becomes a categorical feature with 15 indicators for the codes in the training table and one for a missing value. The model thus sees 35 numerical and 44 categorical attributes.

4.7.2.3 Fitting on the Training Rows

Each preprocessing step uses statistics of the data: the means and standard deviations of the numerical features, and the set of categories that become columns. Section 1.2 computed such statistics from the whole table and warned that the shortcut should not be copied. Here we keep the promise. The statistics come from the training rows alone, and the same transformation is applied to all other rows, so that nothing about the rows on which we measure

the error enters the model before it is evaluated. The error estimate would otherwise be too optimistic, a defect called *leakage*. The rule has a consequence for the encoding: a category that does not occur in the training rows gets no column, so a house with such a value has all of that feature's indicators equal to 0. The `MSSubClass` code 150, for example, occurs in the test table but not in the training table.

The function `preprocess` fits the transformation on the table `train` and applies it to `train` and to any further tables. It returns NumPy arrays of features, without the identifier and the price.

```
def preprocess(train, *others):
    """Fit the feature preprocessing on train; apply it to all tables."""
    frames = [df.drop(columns=['Id', 'SalePrice'], errors='ignore')
                .astype({'MSSubClass': str})  # A code, not a quantity
              for df in (train, *others)]
    numeric = frames[0].select_dtypes(include='number').columns
    mean, std = frames[0][numeric].mean(), frames[0][numeric].std()
    for df in frames:
        df[numeric] = ((df[numeric] - mean) / std).fillna(0)
    frames = [pd.get_dummies(df, dummy_na=True) for df in frames]
    return [df.reindex(columns=frames[0].columns, fill_value=0)
              .to_numpy(dtype=np.float32) for df in frames]
```

Fitted on the whole training table, the encoding turns the 79 attributes into 345 features:

```
X_train, X_test = preprocess(raw_train, raw_test)
X_train.shape, X_test.shape
```

```
((1460, 345), (1459, 345))
```

The label needs one step of preparation as well. The logarithms of the prices lie around 12, with a standard deviation of 0.4. A model whose initial predictions are near 0 would spend its first updates learning this offset, which tells us nothing about any particular house (exercise 3 measures what this costs). We therefore subtract the mean log price of the training rows from the targets, and add it back to the predictions. Shifting targets and predictions by the same constant leaves their differences, and hence the RMSE, unchanged.

The data module `KaggleHouse` holds one split of the labeled houses into training and validation rows. It fits the preprocessing and the mean log price on the training rows, and it serves the features together with the centered log prices. A table without prices, such as the test table, can take the place of the validation rows: the module then provides its features for prediction but no validation data.

```
class KaggleHouse(d2l.DataModule):
    """Features and centered log prices, fitted on the training rows."""
    def __init__(self, raw_train, raw_val, batch_size=64):
        super().__init__()
        self.save_hyperparameters(ignore=['raw_train', 'raw_val'])
        self.X_train, self.X_val = preprocess(raw_train, raw_val)
        self.num_inputs = self.X_train.shape[1]
        log_price = lambda df: np.log(
```

```
            df['SalePrice'].to_numpy(np.float32)).reshape(-1, 1)
        self.offset = log_price(raw_train).mean()  # Mean log price
        self.y_train = log_price(raw_train) - self.offset
        self.y_val = (log_price(raw_val) - self.offset
                      if 'SalePrice' in raw_val else None)

    def get_dataloader(self, train):
        X, y = ((self.X_train, self.y_train) if train
                else (self.X_val, self.y_val))
        if y is None:  # No prices to validate against
            return None
        return self.get_tensorloader((d2l.tensor(X), d2l.tensor(y)), train)
```

4.7.3 Model Selection

4.7.3.1 K-Fold Cross-Validation

To choose between models we need an estimate of their error on new houses. A single validation set would be small: if we held out a fifth of the 1,460 houses, the estimate would rest on 292 of them, and it would depend on which ones. We therefore use the K-fold cross-validation of Section 2.6, in which every house serves for validation once and the average of the K validation errors is our estimate. (In this section K counts folds, not layers.) We use $K = 5$.

`k_fold_data` draws the folds and builds one `KaggleHouse` per round, so that the preprocessing of each round is fitted on its own training folds. Fitting it once on all labeled houses would be simpler, and it would leak the statistics of each validation fold into its features.

```
def k_fold_data(raw, k):
    """Split the table raw into k random folds; one data module per round."""
    folds = np.array_split(np.random.default_rng().permutation(len(raw)), k)
    return [KaggleHouse(raw.drop(index=raw.index[f]), raw.iloc[f])
            for f in folds]
```

`k_fold` trains a fresh model in each round and prints the training and validation RMSE averaged over the rounds, together with the smallest and the largest validation RMSE of a single fold. It returns the training and the validation RMSE of every fold. Each of them is computed from the mean squared error of log prices, averaged over the last five epochs in the trainer's history; its square root is the RMSE of the competition. We average for the reason we averaged accuracies in Section 4.2: a single epoch can land far off. This table gives minibatch SGD a particular opportunity. Every house has 45 features equal to 1, the bias and one indicator per categorical attribute, so the predictions as a whole move in larger steps than any single weight, and now and then an epoch ends on such a step.

The argument `make_model` builds a model for a given number of input features. This number varies slightly from round to round, between about 335 and 345, because each round fits its own set of categories. Only the JAX models need it: NNX layers take their number of inputs when they are constructed, whereas the layers of the other frameworks infer it from the first minibatch.

```
def k_fold(folds, make_model, max_epochs=100):
    """Train a fresh model in each round; return the RMSEs of every fold."""
    rmse = []
    for fold in folds:
        trainer = d2l.Trainer(max_epochs=max_epochs)
        trainer.board.display = False
        trainer.fit(make_model(fold.num_inputs), fold)
        rmse.append([np.mean(trainer.history[f'{split}_loss'][-5:]) ** 0.5
                     for split in ('train', 'val')])  # Last five epochs
    train, val = np.array(rmse).T
    print(f'training RMSE {train.mean():.3f}, validation RMSE '
          f'{val.mean():.3f} (folds {val.min():.3f} to {val.max():.3f})')
    return train, val
```

We draw the folds once and evaluate every model on the same folds, so that differences between the models do not come from different splits.

```
folds = k_fold_data(raw_train, 5)
```

4.7.3.2 A Linear Baseline

The first model is linear regression, `LinearRegression` of Section 2.5, with weights drawn from $\mathcal{N}(0, 0.01^2)$ and a zero bias. We train it, and every later model, with minibatches of 64 houses at learning rate 0.02 for 100 epochs. A baseline is only useful if it is trained as carefully as the models it is compared with, so it gets the same budget.

```
make_linear = lambda num_inputs: d2l.LinearRegression(lr=0.02)
```

```
linear = k_fold(folds, make_linear)
```

```
training RMSE 0.113, validation RMSE 0.148 (folds 0.114 to 0.183)
```

The linear model predicts the validation folds with an RMSE of about 0.15, a typical error of about 15% of the price, against about 0.11 on its training folds. (The training figure is a running mean over the minibatches of the last five epochs, not the result of a separate pass.) Single folds differ a lot: within one run, the best fold has an RMSE of 0.11 to 0.13 and the worst one of about 0.2, depending on which houses they contain. The mean over the five folds varies far less, by about a hundredth from one random split to the next. That is still as much as the models differ. Luckily, models evaluated on the same folds share much of this variation, so the difference between two of them is measured more precisely than either error alone.

4.7.3.3 An MLP with Weight Decay

Can an MLP do better? Ours has one hidden layer of 64 ReLU units and one output, He-initialized weights (Section 4.4.2.2), and zero biases. It reuses the squared loss of `LinearRegression`. With about 340 features it has about 22,000 parameters, nineteen times as many as there are training houses in a round, so we expect it to fit the training folds closely, including their noise. Weight decay with strength `wd` (Section 2.7) limits this; as there, we decay

the weights but not the biases. Dropout (Section 4.6) would be the other candidate, and one of the exercises tries it.

```
class HouseMLP(d2l.Module):
    """An MLP with one hidden layer, squared loss, and weight decay."""
    loss = d2l.LinearRegression.loss

    def __init__(self, num_hiddens, wd, lr):
        super().__init__()
        self.save_hyperparameters()
        self.net = nn.Sequential(nn.LazyLinear(num_hiddens), nn.ReLU(),
                                 nn.LazyLinear(1))

    def weight_init(self, module):
        if isinstance(module, nn.Linear):
            nn.init.kaiming_normal_(module.weight, nonlinearity='relu')
            nn.init.zeros_(module.bias)

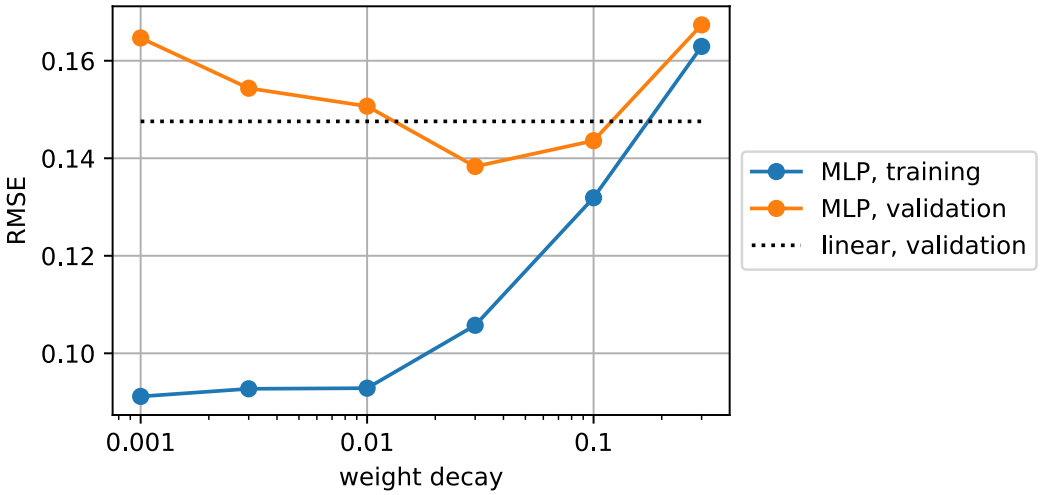
    def configure_optimizers(self):
        weights = [p for p in self.parameters() if p.dim() > 1]
        biases = [p for p in self.parameters() if p.dim() == 1]
        return torch.optim.SGD([{'params': weights, 'weight_decay': self.wd},
                                {'params': biases}], lr=self.lr)

make_mlp = lambda num_inputs, wd: HouseMLP(num_hiddens=64, wd=wd, lr=0.02)
```

We train the MLP with the learning rate, minibatch size, and number of epochs of the linear model, on the same folds, for six strengths of weight decay, from 0.001 to 0.3, each three times the one before, and plot the average training and validation RMSE against the strength, with the validation RMSE of the linear model for reference.

```
wds = [0.001, 0.003, 0.01, 0.03, 0.1, 0.3]
mlp = {}
for wd in wds:
    print(f'weight decay {wd}: ', end='')
    mlp[wd] = k_fold(folds, lambda num_inputs: make_mlp(num_inputs, wd))
d2l.plot(wds, [[mlp[wd][0].mean() for wd in wds],
               [mlp[wd][1].mean() for wd in wds], [linear[1].mean()] * 6],
         'weight decay', 'RMSE', xscale='log', fmts=('C0-o', 'C1-o', 'k:'),
         legend=['MLP, training', 'MLP, validation', 'linear, validation'],
         legend_loc='outside', figsize=(4.5, 3))
```

```
weight decay 0.001: training RMSE 0.091, validation RMSE 0.165 (folds 0.145 to 0.187)
weight decay 0.003: training RMSE 0.093, validation RMSE 0.154 (folds 0.131 to 0.171)
weight decay 0.01: training RMSE 0.093, validation RMSE 0.151 (folds 0.126 to 0.170)
weight decay 0.03: training RMSE 0.106, validation RMSE 0.138 (folds 0.110 to 0.165)
weight decay 0.1: training RMSE 0.132, validation RMSE 0.144 (folds 0.119 to 0.164)
weight decay 0.3: training RMSE 0.163, validation RMSE 0.167 (folds 0.144 to 0.182)
```



The validation curve is the U of Section 2.6, traced here by the strength of weight decay instead of the degree of a polynomial. At its left end, where a decay of 0.001 hardly restrains the weights, the MLP fits the training folds more closely than the linear model did, with a training RMSE of about 0.09 against 0.11, and predicts the validation folds worse, at about 0.16, above the dotted line. It has fit noise. As the decay grows, the training curve rises and the validation curve falls below the dotted line, to its minimum of about 0.135 at a decay of 0.03. At the right end the two curves all but meet, a little below 0.17: the weights are now held so small that the MLP no longer fits even the training folds as well as the linear model does. At the minimum, the MLP is ahead of the linear model by about a hundredth, which lowers the typical error from about 15% to 13% or 14% of the price. For all its 22,000 parameters, the MLP finds little in this table that a weighted sum had missed.

The folds are shared, so we can also compare the best MLP with the linear model fold by fold:

```
best = min(wds, key=lambda wd: mlp[wd][1].mean())
print(f'weight decay {best}: MLP minus linear, per fold',
      np.round(mlp[best][1] - linear[1], 3))
```

```
weight decay 0.03: MLP minus linear, per fold [-0.005  0.013 -0.017 -0.021 -0.016]
```

The MLP is ahead on most folds, by up to about four hundredths, and on the others the two models are within half a hundredth of each other. Its advantage holds up fold by fold, and it is small.

4.7.4 Submitting Predictions

Cross-validation has selected a model: the MLP with the weight decay of the lowest validation RMSE. Each of its five models was trained on four fifths of the houses. For the submission we train it once more, on all 1,460, with the same hyperparameters, as in Section 2.6. The test table takes the place of the validation rows, so the preprocessing and the mean log price are fitted on the whole training table. The model predicts centered log prices; we add the mean log price back, exponentiate, and write one line per house to the file `submission.csv`.

```
data = KaggleHouse(raw_train, raw_test)
model = make_mlp(data.num_inputs, wd=best)
trainer = d2l.Trainer(max_epochs=100)
trainer.board.display = False
trainer.fit(model, data)
log_price = d2l.numpy(model(d2l.tensor(data.X_val))).reshape(-1) + data.offset
submission = pd.DataFrame({'Id': raw_test['Id'],
                           'SalePrice': np.exp(log_price)})
submission.to_csv('submission.csv', index=False)
submission.head()
```

```
     Id      SalePrice
0  1461  122491.210938
1  1462  156303.765625
2  1463  177546.093750
3  1464  195983.781250
4  1465  185933.156250
```

Each line of the file holds the identifier of a test house and its predicted price in dollars, the format that the competition expects. To submit the file, sign in to Kaggle, open the competition page, choose "Submit Prediction", and upload `submission.csv`. Kaggle computes the score and shows it with the position that the submission would take on the leaderboard. Do not expect the score to match the cross-validated RMSE. The model trained on all houses differs from the five models that were evaluated, and the weight decay was selected among six candidates on the same folds that estimate its error, which makes the estimate a little too favorable.

Retraining is not the only way to produce the predictions. The five models of the selected configuration could instead each predict the test houses, and their log-price predictions could be averaged; exercise 6 compares the two.

Nothing in this pipeline is specific to neural networks. The same preprocessing, the same loss, and the same folds serve any model that predicts a number from a vector of features. On tables like this one, ensembles of decision trees trained by gradient boosting often predict better than MLPs (Grinsztajn et al. 2022; Shwartz-Ziv and Armon 2022), and exercise 4 measures how much better on this dataset.

This is a sobering place for a chapter on MLPs to end, and an instructive one. We asked in Chapter 4 for a model in which the evidence from one input depends on the others, and a hidden layer provides it. Of the three problems that hidden layers raised, two are solved: backpropagation computes the gradients, and the right initial scale lets them reach every layer. The third, why a network that could memorize its data generalizes instead, we could measure but not settle. A fully connected layer, however, knows nothing of how its inputs are arranged. To it a house is a list of 345 numbers and an image a list of 784. Neural networks have the advantage where the inputs have structure that a layer can exploit, such as the neighboring pixels of an image or the successive words of a sentence. The book turns there after one chapter on the software that all of its models share.

4.7.5 Summary

A table of mixed numerical and categorical attributes becomes a matrix of features in three steps: standardize the numerical features, replace their missing values by the mean, and

encode the categories, including numbers that only name categories, as indicator features, with one more indicator for a missing value. All statistics of these steps come from the training rows only; otherwise the error estimate leaks information from the rows it is computed on. Prices matter relative to their size, so we predict their logarithms with the squared loss, which trains the model on the square of the competition's score. K-fold cross-validation estimates the error of each candidate from all labeled houses, and evaluating every candidate on the same folds makes their comparison fair. An MLP with 64 hidden units fits the noise of the training houses unless weight decay restrains it. With the weight decay that cross-validation selects, it predicts slightly better than a linear model.

4.7.6 Exercises

1. code **Mean imputation.** `preprocess` replaces every missing numerical value by the training-set mean of its column. Its call to `pd.get_dummies` with `dummy_na=True` also adds a missing-value indicator column for every categorical feature.
    1. Identify a numerical feature in this dataset whose missingness is not random with respect to the sale price.
    2. Explain what bias mean imputation alone would introduce for that feature, and in which direction it would shift the predictions for the affected houses. Then determine whether the indicator columns that `preprocess` already creates let a linear model remove this bias.
    3. Replace mean imputation by median imputation, and separately add a binary missingness indicator column for each imputed feature. Compare the cross-validated errors of the three variants.

    Adapted from Kaggle Learn, Intermediate Machine Learning, Exercise 2.
2. code **Numbers that name categories.** `preprocess` encodes `MSSubClass` as a category because its codes name types of dwelling.
    1. Find the other numerical columns with at most a dozen distinct values, among them `MoSold`, `YrSold`, `OverallQual`, and `OverallCond`. For each, decide whether its values are quantities, ordered levels, or names, and whether one weight on the standardized value can express its effect on the log price. Which of them is cyclic, so that its largest and smallest values are neighbors?
    2. One-hot encode the columns that you judged to be categories, and compare the cross-validated RMSE of the linear model and of the selected MLP with that of this section on the same folds. Is the difference larger than the variation between random splits?
3. code **Scaling the features and the target.** Remove the standardization from `preprocess`. Because `fillna(0)` imputes the column mean only for standardized columns, fill missing numerical values with the training-set column means explicitly instead.
    1. Rerun the cross-validation of the linear model at the section's learning rate of 0.02, and describe the training loss.
    2. Find, to within a factor of ten, the largest learning rate at which the unstandardized linear model trains without diverging. Compare its cross-validated error after 100 epochs with that of the standardized pipeline, and explain both observations.
    3. Restore the standardization, but train on the log prices without subtracting their mean. Compare the cross-validated error after 100 epochs with that of the centered targets, and explain the difference.
    4. With the section's pipeline, record the validation RMSE of the linear model and the mean of its residuals on the validation fold after every epoch. In the epochs

where the RMSE jumps, what has moved? Every house has 45 features equal to 1. Explain why a step of SGD changes the mean prediction more than it changes any single weight, and check whether halving the learning rate removes the jumps.

4. code **Gradient-boosted trees.** On tabular data like this, ensembles of decision trees often predict better than neural networks (Grinsztajn et al. 2022; Shwartz-Ziv and Armon 2022). Fit a gradient-boosted tree model (for example scikit-learn's `GradientBoostingRegressor`, or XGBoost or LightGBM if installed) to the centered log prices of the training rows of each fold in `folds`, and compute its validation RMSE on each fold. Compare the average with those of this section's models. Which properties of the data favor trees?
5. code **Target leakage.** A feature that is recorded only after, or because of, the event being predicted inflates validation scores without helping a deployed model.
   1. Identify at least one raw feature in this dataset that could plausibly be filled in only after the sale, and explain why cross-validation does not catch the problem.
   2. Verify the mechanism empirically: add a synthetic feature equal to the log sale price plus small Gaussian noise, rerun the cross-validation, and report the change in the validation RMSE.

   Adapted from Kaggle Learn, Intermediate Machine Learning, Exercise 7.
6. code **Fold ensembling versus retraining.** Section 4.7.4 retrains the selected MLP on all houses. Alternatively, the five models trained during cross-validation could each predict the test houses, with the preprocessing of their own round, and their log-price predictions could be averaged.
   1. Implement this fold ensemble and submit both its predictions and those of the retrained model. Compare the two leaderboard scores with each other and with the cross-validated RMSE.
   2. Each model of the ensemble was trained on four fifths of the houses. Why might the ensemble nevertheless predict better than the model retrained on all of them?
   3. Explain why the training error of the retrained model is not a substitute for the cross-validated estimate.
7. code **Target encoding.** ▯ *Target encoding* replaces each category of a categorical column by the mean log sale price of the training rows in that category, so that the feature occupies a single column instead of the one column per category that `preprocess` creates.
   1. Target-encode every categorical column with more than ten categories instead of one-hot encoding it, computing the category means from the training rows of each fold. Give a category that does not occur in those rows the overall mean, and standardize the encoded columns like the numerical features. Compare the cross-validated error with that of one-hot encoding.
   2. The encoding above computes each training row's value from a mean that includes that row's own label. Encode the training rows out of fold instead: split the training rows of each fold into five parts, and encode each part with category means computed from the other four parts. Compare the cross-validated error with that of the in-sample encoding, and explain how the size of a category affects the difference between the two encodings.
8. code **Improving the score.** ▯ Lower the cross-validated RMSE of the MLP, and then its leaderboard score.
   1. Vary the width and the number of hidden layers, the learning rate, and the number of epochs, and tune the weight decay of each configuration on the same folds. Which change helps most?

2. Replace weight decay by dropout (Section 4.6) after the hidden layer, and tune its probability. Does dropout help on this dataset, alone or together with weight decay?
3. You have now compared many configurations on the same five folds. Why is the lowest cross-validated RMSE among them a biased estimate of the leaderboard score of the chosen configuration? Check how large the bias is by drawing new folds.

9. **Geometric averaging.** An ensemble of K models that predict the log prices $z_1, \dots, z_K$ of a house can average them before exponentiating, which amounts to a geometric mean in price space.
   1. Compare $\exp\big(\frac{1}{K}\sum_{k=1}^{K} z_k\big)$ with the price-space average $\frac{1}{K}\sum_{k=1}^{K} e^{z_k}$. Show which of the two never exceeds the other, and state when they coincide.
   2. Let $\log y$ be the true log price of the house. Show that the log-space average satisfies

$$\Big(\log y - \frac{1}{K}\sum_{k=1}^{K} z_k\Big)^2 \le \frac{1}{K}\sum_{k=1}^{K} \big(\log y - z_k\big)^2 . \tag{4.32}$$

   Determine whether the logarithm of the price-space average satisfies the same bound for all predictions $z_1, \dots, z_K$.

Chapter 5

Computation

So far each model has been a few lines of code: a class with some layers, a loss, and a call to `fit`. When we trained the multilayer perceptron of Section 4.2, the framework handed its four parameter arrays to the optimizer, kept them on a device of its choosing, and discarded them when the notebook closed. We never had to ask how. Once a model grows, we have to. Take a language model with 8 billion parameters, to which most sections of this chapter return. Its weights alone take 32 GB in the 32-bit format we have used so far. Does the model fit in the memory of a GPU? Can its training resume after a crash? What if it needs a layer that the library lacks?

Early neural network programs derived and coded the gradients of each model by hand. Around 1990 Léon Bottou and Yann LeCun built their network simulator from *modules* instead, each computing its outputs forward and its gradients backward, so that a new architecture is a new arrangement of existing modules. Torch and then PyTorch kept this design, and every framework in this book organizes a model as a tree of modules that own their parameters. The optimizer, the initializer, the checkpoint, and the move to a GPU all walk this tree. This is the one idea to take from the chapter. Knowing the tree lets us change the defaults when they fail, and they do fail: the badly initialized network of Section 4.4 did not train, and a model that is saved incompletely cannot resume its training.

The chapter builds a model, then chooses the numbers it computes in, then runs it. Building takes four sections: how a model is assembled from modules (Section 5.1), what state it holds and how much memory training it needs (Section 5.2), how to set that state before training (Section 5.3, which continues the analysis of Section 4.4), and how to add a layer that the library lacks (Section 5.4). Section 5.5 is about number formats: what a model saves by computing in 16 bits instead of 32, and what it can lose without any error message. The last three sections run the model. We save it and load weights trained by others (Section 5.6), place it on a GPU and account for the memory (Section 5.7), and ask when two runs of the same code give the same result (Section 5.8).

The chapter introduces no new models and no new datasets, which makes it tempting to skip. We advise against that: later chapters rely on it throughout. Parts of it are reference material, however, and the sections say which of their parts can be read quickly and consulted later. The chapter covers one device at a time; training on several devices follows in Chapter 13.

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.

Books

- Deep Learning with PyTorch — Stevens, Antiga & Viehmann — free PDF from the PyTorch team (archived copy; the original pytorch.org link has gone away); Part 1 covers tensors, storage, autograd, and `nn.Module` at the same level as Section 5.1 and Section 5.2.
- Machine Learning Systems — Vijay Janapa Reddi — free online; the systems view around this chapter: frameworks, data pipelines, training infrastructure, and efficient deployment (Section 5.7, Section 5.8).

Courses and video lectures

- CMU 10-414/714: Deep Learning Systems — Chen & Kolter — free lectures and assignments; students build "needle", a miniature framework with autograd, modules, initialization, and GPU support, an implementer's view of Section 5.1, Section 5.3, and Section 5.7.
- Neural Networks: Zero to Hero — Andrej Karpathy — free video series; builds autograd, modules, and training loops from scratch in plain Python, the abstractions behind Section 5.1 and Section 5.4.
- fast.ai Part 2: Deep Learning Foundations — Howard et al. — free; rebuilds a training framework from tensor operations upward, including modules, initialization, and mixed precision (Section 5.1, Section 5.3, Section 5.5).

Tutorials, notes, and interactive

- PyTorch internals — Edward Yang — free; a guided tour of tensors, strides, dispatch, and autograd inside PyTorch, one level below Section 5.1.
- JAX — The Sharp Bits — free; pure functions, explicit PRNG keys, and jit constraints — the functional worldview behind this book's JAX tab, stated as a list of gotchas (Section 5.2, Section 5.8).
- Train With Mixed Precision — NVIDIA — free; the vendor guide to fp16/bf16 arithmetic and loss scaling that Section 5.5 distills.
- What Every Computer Scientist Should Know About Floating-Point Arithmetic — David Goldberg — free; the standard reference beneath every rounding and overflow issue in Section 5.5.
- Making Deep Learning Go Brrrr From First Principles — Horace He — free; a first-principles account of compute, memory, and framework overhead covered in Section 5.7.
- safetensors — Hugging Face — free; the file format for sharing weights without executing code on load, and why it replaced pickle-based checkpoints (Section 5.6).
- Reproducibility — PyTorch notes — free; the determinism flags, their costs, and their limits, complementing Section 5.8.

Foundational papers

- Automatic Differentiation in Machine Learning: a Survey — Baydin, Pearlmutter, Radul & Siskind (2018), *JMLR* — free; the definitive account of forward- and reverse-mode autodiff, the algorithm behind every custom gradient in Section 5.4.

- Mixed Precision Training — Micikevicius et al. (2018) — free; the origin of the fp16 + master-weights + loss-scaling recipe in Section 5.5.
- PyTorch: An Imperative Style, High-Performance Deep Learning Library — Paszke et al. (2019) — free; the design rationale (eager execution, autograd, memory allocator) for the imperative style of Section 5.1 and Section 5.7.

5.1 Modules and Model Construction

The multilayer perceptron of Section 4.2 had four parameter arrays, and we could name each one by hand. The models in the rest of this book have many more. A residual network for images stacks dozens of layers, and the language model Llama 3 8B (Grattafiori et al. 2024), with 8 billion parameters, consists of 32 identical blocks and stores its parameters in 291 arrays. Every one of them must reach the optimizer, be written to a checkpoint, and move to the GPU. An array that is left out raises no error: it is simply never trained, or never saved.

Deep learning frameworks solve this with one abstraction, the *module*. A module is an object that owns parameters, owns other modules (its *children*), and defines a forward computation from inputs to outputs. Layers, groups of layers, and whole models are all modules, so a model is a tree of modules, as in Figure 5.1. The framework finds every parameter by walking this tree, provided that each child has been *registered* with its parent. This section is a practical one, and a reader who has written modules before will move through much of it quickly. It shows how registration works and how it fails, which is the part not to skip, how to write the forward computation of a module ourselves, and how to describe a whole model by a handful of numbers. The rest of the chapter uses the same walk for other tasks: counting the parameters (Section 5.2), initializing them (Section 5.3), saving them (Section 5.6), and moving them to a GPU (Section 5.7).

```
from dataclasses import dataclass
```

5.1.1 The Module Abstraction

5.1.1.1 Sequential Containers and the Registry

Every model since Section 2.4 has been a module: `d2l.Module` is a subclass of the framework's module class. The quickest way to build a module from layers is a *sequential container*, a module that applies its children one after the other. Here it holds the MLP of Section 4.2, with a hidden layer of 256 ReLU units and 10 outputs, applied to a minibatch of two examples with 20 features each.

In PyTorch the module class is `nn.Module`, and the container `nn.Sequential` takes its layers as arguments. Each `nn.Linear` receives its number of inputs and of outputs.

```
net = nn.Sequential(nn.Linear(20, 256), nn.ReLU(), nn.Linear(256, 10))

X = torch.rand(2, 20)
net(X).shape
```

```
torch.Size([2, 10])
```

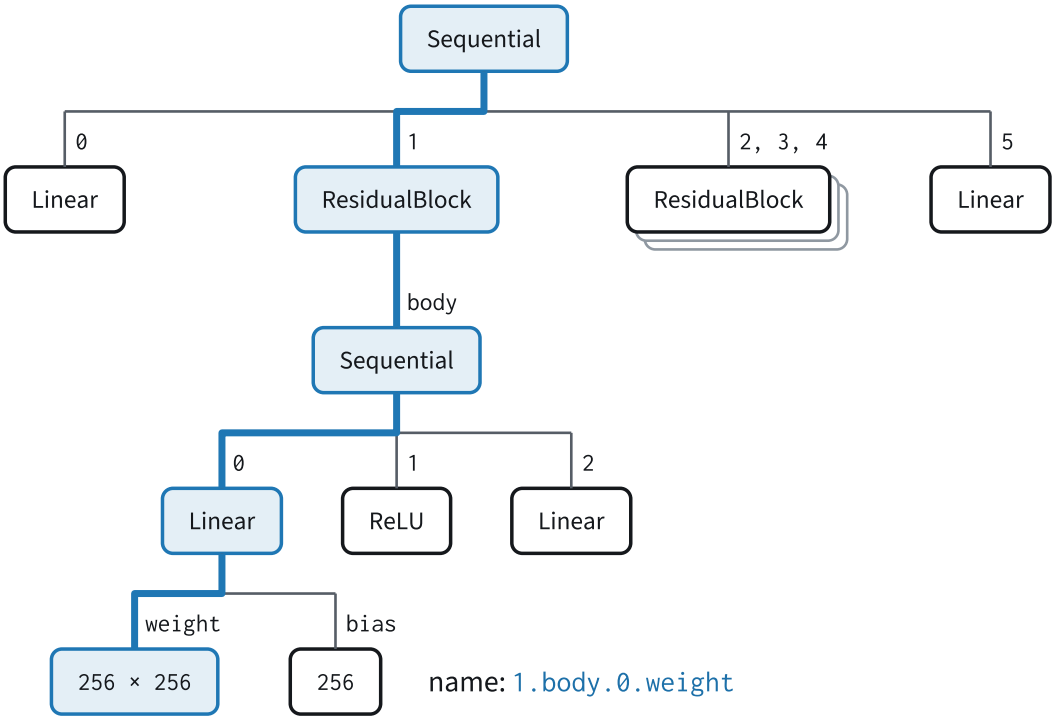


Figure 5.1: The module tree of the residual MLP that we build at the end of this section. Each edge carries the name under which the parent registered the child, and the path of names from the root to a parameter is the name of that parameter, here `1.body.0.weight`. Blocks 2 to 4 repeat block 1.

The container keeps its children in a *registry*, a table from names to modules, and its forward computation runs through that table in order.

`named_children` lists the registry. The names are the positions `'0'`, `'1'`, and `'2'`; the ReLU is a module without parameters.

```
dict(net.named_children())
```

```
{'0': Linear(in_features=20, out_features=256, bias=True),
 '1': ReLU(),
 '2': Linear(in_features=256, out_features=10, bias=True)}
```

This table is where every walk over the model starts. Collecting the parameters for the optimizer, writing them to a file, and moving them to a GPU all visit the registered children, then their children, and so on down to the parameters. For all three, a layer that is not in the table does not exist.

5.1.1.2 Subclasses of the Module Class

A sequential container only expresses a chain. For any other wiring we subclass the module class and write two methods: a constructor that creates the children, and a forward method that applies them to the input. The forward method is called `forward` in PyTorch and Gluon, `call` in Keras, and `__call__` in NNX.

Here is the same MLP once more:

```
class MLP(nn.Module):
    def __init__(self):
        super().__init__()
        self.hidden = nn.Linear(20, 256)
        self.out = nn.Linear(256, 10)

    def forward(self, X):
        return self.out(F.relu(self.hidden(X)))
```

```
net = MLP()
net(X).shape
```

```
torch.Size([2, 10])
```

Two details make this work. First, `self.hidden = ...` is not an ordinary attribute assignment. The module class intercepts assignment (the Python method `__setattr__`), and when the value is a module it enters the registry under the attribute name, here `hidden` and `out`. That is all it takes for both layers' parameters to reach the optimizer. Second, we wrote no backward method: automatic differentiation derives the gradients from whatever the forward method computes.

We call the model as `net(X)`, never as `net.forward(X)`: `nn.Module.__call__` runs `forward` and, around it, any *hooks* registered on the module, functions that see its inputs and outputs (Section 5.8).

5.1.2 Module Containers

5.1.2.1 Writing a Container

A container has two jobs: to register each child, and to run the children in order. It is instructive to write one ourselves, since the first job is the one that can go wrong.

```
class MySequential(nn.Module):
    def __init__(self, *args):
        super().__init__()
        for idx, module in enumerate(args):
            self.add_module(str(idx), module)

    def forward(self, X):
        for module in self.children():
            X = module(X)
        return X
```

`add_module` enters a child into the registry under a name, here its position as a string, and `children()` runs through the registry in the order of insertion.

Our container works like the built-in one:

```
net = MySequential(nn.Linear(20, 256), nn.ReLU(), nn.Linear(256, 10))
net(X).shape
```

```
torch.Size([2, 10])
```

5.1.2.2 Layers in a Plain List

Now suppose that we store the layers in an ordinary Python list, as anyone would who had not read the previous subsection. The forward method can still loop over the list, but the list itself is not a module. Does the framework still find the layers inside it?

```
class PlainListMLP(nn.Module):
    def __init__(self):
        super().__init__()
        self.layers = [nn.Linear(20, 256), nn.ReLU(), nn.Linear(256, 10)]

    def forward(self, X):
        for layer in self.layers:
            X = layer(X)
        return X

net = PlainListMLP()
net(X).shape, sum(p.numel() for p in net.parameters())
```

```
(torch.Size([2, 10]), 0)
```

The model computes, yet it owns zero parameters. A plain Python list is not a module, so `__setattr__` stores it like any other value and registers nothing inside it. The layers still hold their own weights, which is why the forward pass runs. But `net.parameters()` is empty, `net.to(device)` leaves every layer on the CPU, and `net.state_dict()` saves an empty model.

How would we notice? Here the mistake is loud: `torch.optim.SGD` refuses an empty list of parameters, so this model cannot even be handed to an optimizer. The dangerous model is the one that is only partly registered. In the model below the outer layers are attributes, and the two layers between them sit in a plain list. We take one step of gradient descent and ask which weights changed:

```
class PartlyRegisteredMLP(nn.Module):
    def __init__(self):
        super().__init__()
        self.first = nn.Linear(20, 256)
        self.blocks = [nn.Linear(256, 256), nn.Linear(256, 256)]  # Plain list
        self.last = nn.Linear(256, 10)

    def forward(self, X):
        X = F.relu(self.first(X))
        for block in self.blocks:
            X = F.relu(block(X))
        return self.last(X)
```

```
net = PartlyRegisteredMLP()
layers = [net.first, *net.blocks, net.last]
before = [layer.weight.clone() for layer in layers]
optimizer = torch.optim.SGD(net.parameters(), lr=0.1)
net(X).sum().backward()
optimizer.step()
[not torch.equal(b, layer.weight) for b, layer in zip(before, layers)]
```

```
[True, False, False, True]
```

Only the weights of the outer layers changed, and nothing failed. Backpropagation passed through the two layers in the list to reach the first one, but the optimizer never received their parameters. On the CPU the only sign is a model that learns less than it should, which is the kind of bug that survives for months. On a GPU the mistake is loud again: `Trainer(num_gpus=1)` moves only the registered layers, and the first forward pass stops with an error about tensors on different devices.

The registered container for a list of children is `nn.ModuleList`:

```
class ModuleListMLP(PlainListMLP):
    def __init__(self):
        super().__init__()
        self.layers = nn.ModuleList(self.layers)

net = ModuleListMLP()
net(X).shape, sum(p.numel() for p in net.parameters())
```

```
(torch.Size([2, 10]), 7946)
```

Same forward pass, and now all 7,946 parameters are registered: $20 \cdot 256 + 256$ in the hidden layer and $256 \cdot 10 + 10$ in the output layer.

Which container, then? `nn.Sequential` registers its children and supplies a forward method

that chains them. `nn.ModuleList` registers a list of children and supplies no forward method, so we write the loop ourselves, which is what we want when the loop is not a plain chain: blocks that take an extra argument, or a computation between blocks. `nn.ModuleDict` does the same for children indexed by name. A model that repeats one block many times, such as the 32-block 8B model of the introduction, keeps the blocks in an `nn.ModuleList`.

5.1.3 Forward Computation

The forward method is ordinary Python. It can call any tensor operation, loop, branch, and combine intermediate results. We use this freedom for a residual connection, and then ask what a compiled training step takes away from it.

5.1.3.1 Residual Connections

Let us start with the simplest module whose forward method is not a chain. A *residual connection* adds the input of a block to its output [He et al. (2016)*1]. If f is the *body* of the block, the computation inside it, here two fully connected layers with a ReLU between them, the block computes

$$\mathbf{y} = \mathbf{x} + f(\mathbf{x}). \tag{5.1}$$

If f is zero, the block is the identity, so a stack of such blocks can pass its input through unchanged, and each block only has to learn a correction $f(\mathbf{x})$ to its input. The networks of Section 7.4 are built this way. No framework provides the residual connection as a layer, and none needs to. It is one line of the forward method, `X + self.body(X)`:

```
class ResidualBlock(nn.Module):
    def __init__(self, num_hiddens):
        super().__init__()
        self.body = nn.Sequential(
            nn.Linear(num_hiddens, num_hiddens), nn.ReLU(),
            nn.Linear(num_hiddens, num_hiddens))

    def forward(self, X):
        return X + self.body(X)
```

Figure 5.2 shows the wiring. The addition has one consequence for construction: $\mathbf{x}$ and $f(\mathbf{x})$ must have the same shape, so a residual block has a single width, which we pass to the constructor.

```
block = ResidualBlock(24)
block(torch.randn(2, 24)).shape
```

```
torch.Size([2, 24])
```

A forward method may also read values that are neither inputs nor parameters. A fixed number can stay a plain Python attribute. An array computed from data, such as the mean of the inputs, is another matter: it must be saved and moved with the model, yet never trained. Section 5.2.1.2 shows how to register such state.

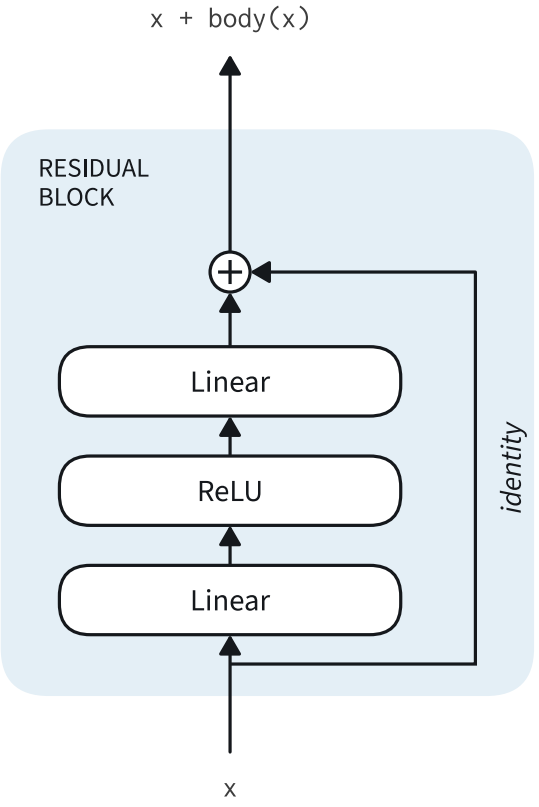


Figure 5.2: A residual block. The input splits into the body, Linear, ReLU, Linear, and an identity path, and the two rejoin by addition.

5.1.3.2 Eager and Compiled Execution

So far the forward method has also run like ordinary Python, one line at a time. A framework can instead record the computation once and replay the recording. This is faster, but it restricts what the forward method may do.

PyTorch runs `forward` as written, one operation at a time, a mode called *eager execution*, and so does our `Trainer`. A loop whose condition depends on the values of tensors therefore works just as it does in NumPy. Compilers for PyTorch models exist, and Chapter 13 uses them.

5.1.4 Lazy Initialization

5.1.4.1 Shapes from Data

Since Section 2.5, our PyTorch models have used lazy layers such as `nn.LazyLinear(256)`, which name only their output width. The weight has shape `(256, in_features)`, and we never said what `in_features` is: the layer reads it from the first minibatch. Here is the mechanism. Before the first forward pass the weight is a placeholder without a shape:

```
net = nn.Sequential(nn.LazyLinear(256), nn.ReLU(), nn.LazyLinear(10))
net[0].weight
```

```
<UninitializedParameter>
```

The first forward pass reads the input width from the minibatch, creates the weight, and turns the layer into an ordinary `nn.Linear`. The output width of each layer is the input width of the next, so the shapes cascade through the model:

```
net(X)
net[0].weight.shape
```

```
torch.Size([256, 20])
```

5.1.4.2 Initialization after a Dry Run

Lazy layers spare us the arithmetic of input widths, which in a deep model is tedious and easy to get wrong. Unfortunately, the parameters then do not exist until a minibatch has passed through. Before that, they cannot be counted, and an initializer whose scale depends on the number of inputs, such as the Xavier initialization of Section 4.4.2.1, cannot be applied. Initialization therefore follows a *dry run*, one forward pass on a representative minibatch whose output is discarded.

The `Trainer` has done this before every training run since Section 2.5, whenever the model still had lazy parameters: a dry run, then the model's `weight_init`. The method behind it, `apply_init` of `d2l.Module`, we have not shown so far. It runs the model once without recording gradients, then calls the given function on every submodule:

```
@d2l.add_to_class(d2l.Module)
def apply_init(self, inputs, init=None):
    """Run a dry pass to fix lazy shapes, then apply `init`."""
```

```
    with torch.no_grad():
        self(*inputs)
    if init is not None:
        self.apply(init)
```

`nn.Module.apply(fn)` calls `fn` on every module of the tree, children before their parents: one more task that is a walk over the tree. Here is `apply_init` with Xavier's uniform initializer:

```
class TinyMLP(d2l.Module):
    def __init__(self):
        super().__init__()
        self.net = nn.Sequential(nn.LazyLinear(256), nn.ReLU(),
                                 nn.LazyLinear(10))

def init_xavier(module):
    if isinstance(module, nn.Linear):
        nn.init.xavier_uniform_(module.weight)

model = TinyMLP()
model.apply_init([X], init_xavier)
model.net[0].weight.shape
```

```
torch.Size([256, 20])
```

The dry run turned every lazy layer into an `nn.Linear`, after which `init_xavier` could recognize the layers and redraw their weights.

Which should we prefer? For a quick experiment on new data, lazy layers are a convenience. A model whose parameters are to be counted, initialized by a rule, or built from a config should state both widths of every layer, as the models in the rest of this chapter do.

5.1.5 Models from a Config

5.1.5.1 Configs and Builders

Could we rebuild a model from its saved weights alone? Not quite. A trained model is a set of arrays plus the code that knows what to do with them. Every width and depth in this section so far was a literal typed into a constructor call, and a checkpoint of the weights records such choices at best indirectly: the shapes of the arrays reveal the widths, but nothing in them reveals a choice without parameters, such as the activation function.

The choices that vary from one experiment to the next therefore belong in a small *configuration* object, or *config*, that is logged with the results and saved with the weights. The topology, meaning what a block computes and how blocks connect, stays in code. Here a Python `dataclass` holds four fields, and `build` turns them into a model:

```
@dataclass
class MLPConfig:
    num_inputs: int = 784
    num_hiddens: int = 256
    num_blocks: int = 4
```

```
    num_outputs: int = 10

def build(cfg: MLPConfig) -> nn.Module:
    blocks = [ResidualBlock(cfg.num_hiddens) for _ in range(cfg.num_blocks)]
    return nn.Sequential(nn.Linear(cfg.num_inputs, cfg.num_hiddens),
                         *blocks, nn.Linear(cfg.num_hiddens, cfg.num_outputs))
```

One config gives one architecture: an input layer from `num_inputs` features to the hidden width, `num_blocks` residual blocks, and an output layer. The blocks pass through a plain Python list, of all things. This is safe here, because unpacking the list into the sequential container registers each block. Every layer receives both of its widths, so the model needs no dry run before its parameters can be counted.

The names of the first parameters are the paths of Figure 5.1, joined by dots:

```
net = build(MLPConfig())
[(name, tuple(p.shape)) for name, p in net.named_parameters()][:4]
```

```
[('0.weight', (256, 784)),
 ('0.bias', (256,)),
 ('1.body.0.weight', (256, 256)),
 ('1.body.0.bias', (256,))]
```

The model maps a minibatch of 784-dimensional inputs, the size of a flattened Fashion-MNIST image, to 10 outputs:

```
net(torch.rand(2, 784)).shape
```

```
torch.Size([2, 10])
```

5.1.5.2 Counting Parameters

The config also tells us the size of the model before we build it. The input layer has $784 \cdot 256 + 256 = 200{,}960$ parameters, each block has two layers of $256 \cdot 256 + 256$, or $131{,}584$ together, and the output layer has $256 \cdot 10 + 10 = 2{,}570$. The default model therefore has

$$200{,}960 + 4 \cdot 131{,}584 + 2{,}570 = 729{,}866 \tag{5.2}$$

parameters, most of them in the blocks. Rescaling the model is a change to two fields, not an edit of the model code, and a walk over the parameters confirms the count:

```
for cfg in (MLPConfig(), MLPConfig(num_hiddens=512, num_blocks=8)):
    n = sum(p.numel() for p in build(cfg).parameters())
    print(f'num_hiddens={cfg.num_hiddens}, num_blocks={cfg.num_blocks}: '
          f'{n:,} parameters')
```

```
num_hiddens=256, num_blocks=4: 729,866 parameters
num_hiddens=512, num_blocks=8: 4,609,546 parameters
```

Doubling the width multiplies the parameters of each block by about four, and doubling the number of blocks doubles their total, so the larger config has more than six times as

many parameters. The 8B model of the introduction is described the same way: a config of a dozen numbers, such as 32 blocks and a width of 4,096, and code that turns it into a module tree.

Note, however, that a config records numbers, not code. Rebuilding a model from a checkpoint needs the weights, the config, and the same `build` function and block classes. Section 5.6 saves the first two together in one file; the code we must keep ourselves.

Throughout this section the parameters were things to be found and counted. We have not yet looked at them: which tensors a model holds, how much memory they take, and how to share or freeze them. This is the subject of Section 5.2.

5.1.6 Summary

Three things are worth remembering. A model is a tree of modules, and the name of each parameter is its path from the root. The framework finds parameters by walking the registered children, so a child stored where the walk cannot see it is never trained or saved, and in the worst case nothing tells us. A config of a few numbers, together with a `build` function, records an architecture so that it can be logged, saved, and rebuilt. The rest is technique: a forward method written in ordinary Python, which a compiled training step restricts, and lazy layers, which must see a minibatch before they can be initialized.

In PyTorch, children are registered by attribute assignment and by the containers `nn.Sequential`, `nn.ModuleList`, and `nn.ModuleDict`. Modules in a plain list stay unregistered, without any warning.

5.1.7 Exercises

1. code **Unregistered modules.** The plain list in `PlainListMLP` keeps its layers out of `net.parameters()`, `net.to(device)`, and `net.state_dict()`. Predict, then verify, what three further operations do to the model.
    1. Call `net.apply(init_xavier)` and check whether the weight of `net.layers[0]` changes.
    2. Add `nn.Dropout(0.5)` after the `nn.ReLU()` in the list, call `net.eval()`, and pass the same input through the model twice.
    3. The method `load_state_dict` is the inverse of `state_dict`: it copies a dictionary of saved tensors into a model. Call `net.load_state_dict(ModuleListMLP().state_dict())`.
    4. Which of these failures are silent and which raise an error? Explain all three from the same missing registration.
2. **Widening a residual block.** `ResidualBlock` requires its input and output widths to agree. Suppose a block maps width d to a larger width d', so its body ends in a d'-wide layer and the skip path must also produce width d'. Give one way to build that skip path with parameters and one without. For each, count the extra parameters and state which output coordinates still receive the identity path.
3. code **Config boundaries.** Extend `MLPConfig` with a field `act: str = 'relu'` that selects the activation inside every `ResidualBlock`, and make `build` honor it.
    1. Build one model with `act='tanh'` and one from the default config, and compare their parameter shapes. Could a checkpoint that holds only the weights tell the two models apart? State what the answer implies for the config.

2. A plain feed-forward block has the same `body` as `ResidualBlock` but no skip connection. Apply the same test to the choice between the two block types, and decide whether that choice belongs in the config or in code.

4. code **A module from scratch.** ▯ In plain Python and NumPy, without any framework base class, write a `Parameter` class that wraps an array and a minimal `Module` class whose `__setattr__` registers every assigned `Parameter` or `Module`. Give `Module` a `parameters()` method that recurses into the registered children. Write a `Linear` subclass of `Module` that holds a weight and a bias, build a two-layer MLP from two instances of it, and verify that `parameters()` returns all four arrays. Then store the two layers in a plain list, check what `parameters()` returns, and extend `Module` so that the list version finds all four arrays as well.

   Adapted from CMU 10-714, Homework 2, Question 2.

5. code **Lazy input width.** Build `net = nn.Sequential(nn.LazyLinear(256), nn.ReLU(), nn.LazyLinear(10))` and pass a batch with 20 features through it, then a batch with 30 features. Explain the outcome from what the first forward pass did to each layer. Then describe a model that accepts inputs of either width while sharing every layer after the first.

5.2 Parameters, State, and Memory

The previous section built models as trees of modules and found every parameter by walking the tree. So far the framework has handled those tensors for us, and we never had to ask what else a model holds or how much memory training it takes. For a small model there is no need to ask. For a large one both questions become pressing. Consider a model with 8 billion parameters, the size of the Llama 3 model that Section 5.1 mentioned. Its weights take 32 GB in the 32-bit floating-point format we have used so far. Training it takes four times as much memory, 128 GB, before it has seen a single example: every weight needs a gradient of the same size, and the Adam optimizer, which we meet in Section 9.5, keeps two more tensors of that size, running averages of the gradients and of their squares. That is more than an 80 GB GPU holds.

To arrive at such numbers we need to know every tensor a model holds and what each one is for. Quite ordinary operations need the same knowledge: a checkpoint must save every tensor the computation depends on, a move to a GPU must take all of them along, and fine-tuning must decide which of them the optimizer may change.

Frameworks make this possible by *registration* (Section 5.1): each layer announces the tensors it owns to the module system, which files them by name in the tree of modules, and the optimizer, the checkpoint, and the device move then all walk the same tree. This section is an inventory of that tree. Its first part asks what a model holds: the tensors the optimizer trains, and those it must leave alone. Its second part asks what they cost, and counts the bytes that training stores per parameter. The third part is closer to a reference. It collects three things that are routinely done to parameters (sharing one between two layers, excluding some from training, and averaging them over time), each of which borrows a model or a method from a later chapter. It can be skimmed now and consulted when those chapters call for it.

```
import torch
from torch import nn
```

5.2.1 The State Tree

5.2.1.1 Access by Path

Our running example is a residual MLP: an input layer maps 20 features to 64, two residual blocks of width 64 follow, and an output layer produces 10 outputs. The blocks are the residual block of Section 5.1, defined again here so that this section stands on its own: each adds the output of a small two-layer network, its *body*, to its input.

```
class ResidualBlock(nn.Module):
    def __init__(self, num_hiddens):
        super().__init__()
        self.body = nn.Sequential(
            nn.Linear(num_hiddens, num_hiddens), nn.ReLU(),
            nn.Linear(num_hiddens, num_hiddens))

    def forward(self, X):
        return X + self.body(X)

net = nn.Sequential(nn.Linear(20, 64), ResidualBlock(64),
                    ResidualBlock(64), nn.Linear(64, 10))
X = torch.randn(2, 20)
net(X).shape
```

```
torch.Size([2, 10])
```

The parameters are the leaves of the module tree, and the path of names that leads to a leaf (Figure 5.1) is also the way to reach it in code, which suits debugging a single layer. The optimizer and the checkpoint, however, need every leaf. To this end each framework offers a traversal of the whole tree that returns the parameters together with their paths.

In PyTorch it is `named_parameters()`. Each parameter is an `nn.Parameter`, a tensor that registers itself with the module system as trainable, and its gradient `.grad` stays `None` until backpropagation fills it.

The model has 12 parameter tensors. We print their number and the four that belong to the first residual block:

```
named = list(net.named_parameters())
len(named), [(name, p.shape) for name, p in named[2:6]]
```

```
(12,
 [('1.body.0.weight', torch.Size([64, 64])),
  ('1.body.0.bias', torch.Size([64])),
  ('1.body.2.weight', torch.Size([64, 64])),
  ('1.body.2.bias', torch.Size([64]))])
```

The name `1.body.0.weight` is a path: child `1` of `net` (the first residual block), its submodule `body`, that module's child `0`, and finally the leaf `weight`; in code the same path reads

`net[1].body[0].weight`. These names are also the keys of the model's `state_dict`, the mapping from names to tensors used for saving and loading (Section 5.6):

```
list(net.state_dict()) == [name for name, _ in net.named_parameters()]
```

```
True
```

Every consumer walks this one tree under these names. The equality above holds only because all of this model's state is trainable. That need not be so.

5.2.1.2 Parameters and Buffers

Suppose a model standardizes its inputs with a mean and a standard deviation computed once from training data. These numbers must be saved with the model, since predictions are wrong without them, and they must move to the GPU with it. Yet the optimizer must never change them: they are statistics of the data, not quantities to fit. A plain attribute would not do, since no traversal sees it (Section 5.1). Batch normalization (Ioffe and Szegedy 2015), a layer that we study in Section 7.3, keeps tensors of the same kind: a running mean and variance of its inputs, updated during training and used at prediction time.

Such a persistent tensor without a gradient is commonly called a *buffer*, and every framework has a way to register one so that checkpoints and device moves see it while the optimizer does not. This gives a rule that covers every tensor a layer touches: make it a *parameter* if the optimizer should update it, a *buffer* if checkpoints need it, and a local variable if it lives for one call. The last kind includes the intermediate values that backpropagation stores during a training step (Section 4.3). They belong to the computation, not to the model, and are neither saved nor registered. Plain Python attributes hold architecture constants such as layer widths.

Here is a layer, `Standardizer`, that standardizes its inputs with statistics computed from a reference sample: one mean and one standard deviation per feature.

PyTorch uses the name *buffer* itself and registers one with `register_buffer`.

```
class Standardizer(nn.Module):
    def __init__(self, mean, std):
        super().__init__()
        self.register_buffer('mean', mean)
        self.register_buffer('std', std)
        self.out = nn.Linear(4, 2)

    def forward(self, X):
        return self.out((X - self.mean) / self.std)

sample = torch.randn(100, 4) * torch.arange(1., 5.)
standardizer = Standardizer(sample.mean(0), sample.std(0, correction=0))
list(standardizer.state_dict())
```

```
['mean', 'std', 'out.weight', 'out.bias']
```

The buffers appear in the state dict, so they are checkpointed with the weights. They do not appear among the parameters, so the optimizer never sees them:

```
[name for name, _ in standardizer.named_parameters()]
```

```
['out.weight', 'out.bias']
```

Registration decides more than what is saved. Operations on the model as a whole, such as a change of dtype or a move to another device, reach the registered tensors and no others (Section 5.1). We can watch this without a GPU.

Converting the module to another *dtype* uses the same machinery as moving it to another device. The registered buffer converts; a tensor stored as a plain attribute stays behind:

```
standardizer.note = torch.zeros(4)   # plain attribute, not registered
standardizer.to(torch.float64)
standardizer.mean.dtype, standardizer.note.dtype
```

```
(torch.float64, torch.float32)
```

The same failure appears when moving to a GPU: a model works on the CPU, then crashes after `.to('cuda')` because an unregistered tensor stayed behind. On a machine with a GPU, the following confirms that buffers move with the module:

```
if torch.cuda.is_available():
    print('buffer lives on', standardizer.to('cuda').mean.device)
else:
    print('no GPU here; on a CUDA machine, standardizer.to("cuda") '
          'moves standardizer.mean along with the parameters')
```

```
buffer lives on cuda:0
```

5.2.2 Training Memory

5.2.2.1 Bytes per Parameter

How much memory does training our residual MLP take? For the parameters and what the optimizer stores about them, the answer follows from three counts per parameter: the weight itself, its gradient, and the optimizer's state for it. Minibatch gradient descent, which we have used so far, keeps no state. In 32-bit floating point (fp32), 4 bytes per number, it stores 8 bytes per parameter. Adam keeps two more numbers per parameter, the running averages of the gradient and of its square. We count for Adam, since almost all large models are trained with it or with one of its variants, so this is the count that matters in practice:

Training state	Precision	Bytes per parameter
Weights	fp32	4
Gradients	fp32	4
Adam's average of the gradients	fp32	4
Adam's average of their squares	fp32	4
Total		16

These 16 bytes are a floor. They omit the *activations*, the intermediate values that backpropagation keeps from the forward pass, whose size grows with the batch; they also omit tem-

porary workspaces and the memory the allocator holds in reserve. Section 5.7 measures all of these on a GPU. For our residual MLP, the floor is:

```
n = sum(p.numel() for p in net.parameters())
weights, grads, adam_state = 4 * n, 4 * n, 8 * n   # bytes, all fp32
print(f'{n} parameters: {(weights + grads + adam_state) / 2**20:.2f} MiB '
      'for weights + gradients + Adam state')
```

```
18634 parameters: 0.28 MiB for weights + gradients + Adam state
```

The count checks by hand: the input layer has $20 \times 64 + 64 = 1344$ parameters, each residual block $2 \times (64 \times 64 + 64) = 8320$, and the output layer $64 \times 10 + 10 = 650$, for a total of 18,634. Adam's state is real memory, allocated tensor by tensor. After one training step, its two running averages are two more copies of the model:

```
adam = torch.optim.Adam(net.parameters())
net(X).sum().backward()
adam.step()
moments = sum(t.numel() for state in adam.state.values()
              for t in state.values() if t.ndim > 0)
moments == 2 * n
```

```
True
```

For our network the floor is 0.28 MiB, which is why memory has not mattered so far. For the 8-billion-parameter model the same arithmetic gives the 128 GB of the opening, whatever the batch size. One might hope that computing with 16-bit numbers, the *mixed-precision* training of Section 5.5, halves this figure. Unfortunately it does not lower the floor at all: an fp32 copy of the weights and fp32 running averages are kept, so that small updates are not rounded away (Micikevicius et al. 2018), and the 16-bit copies come on top.

5.2.2.2 Counting Without Allocating

Every count so far read the shapes of tensors that already existed. For the 8B model this order is backwards: to learn whether its 32 GB of weights fit in memory, we would first have to allocate them. Luckily, the count needs only the shape and the dtype of each parameter, and both follow from the arguments of the constructors. Take the output layer of the 8B model, which maps a hidden state of width 4,096 to one score for each of the 128,256 *tokens*, the words and pieces of words, that the model can output. Its weight has $4096 \times 128{,}256 = 525{,}336{,}576$ entries, 2.1 GB in fp32. We create the layer without its data and count.

PyTorch's *meta* device holds tensors that have a shape and a dtype but no data. Inside `with torch.device('meta'):`, a constructor creates its parameters on that device, so the module is complete except for its values:

```
with torch.device('meta'):
    head = nn.Linear(4096, 128256, bias=False)   # shapes, no data
n_head = sum(p.numel() for p in head.parameters())
print(head.weight.device, f'{n_head:,} parameters, '
      f'{4 * n_head / 1e9:.1f} GB in fp32')
```

```
meta 525,336,576 parameters, 2.1 GB in fp32
```

A meta tensor has no storage. To use the model, `to_empty(device=...)` allocates its parameters on a device without filling them, and an initializer or a checkpoint fills them. Large pretrained models are loaded this way, so that each weight is written once, from the file, instead of first being drawn at random and then overwritten. On our residual MLP:

```
with torch.device('meta'):
    clone = nn.Sequential(nn.Linear(20, 64), ResidualBlock(64),
                          ResidualBlock(64), nn.Linear(64, 10))
clone = clone.to_empty(device='cpu')   # memory, no values yet
clone.load_state_dict(net.state_dict())
torch.equal(clone(X), net(X))
```

```
True
```

Applied to every layer of the 8B model, the same count returns its 8 billion parameters, and with them the memory figures of the opening, without allocating a byte.

5.2.3 Shared, Frozen, and Averaged Parameters

5.2.3.1 Tied Parameters

One tensor can serve several roles in a model. The standard example is a language model, which reads a sequence of tokens from a vocabulary V and predicts the next one (Section 8.3). Its first layer is an *embedding*, a $|V| \times d$ table whose row v is the d-dimensional vector that represents token v. Its output layer maps a d-dimensional hidden state to $|V|$ scores, one per token, so its weight matrix has the same shape and an analogous meaning: one vector per token. Why keep two tables? *Tying* the two roles to a single tensor saves $|V| \times d$ parameters, and the studies that introduced it found that it also improves the predictions (Press and Wolf 2017; Inan et al. 2017). This is at least plausible: words that can replace one another in a sentence should be similar as inputs and receive similar scores as outputs, and tying leaves the model only one notion of similarity to learn. The savings are large. In GPT-2 small (Radford et al. 2019), the shared 50257×768 table holds about 38.6 million of the model's 124 million parameters, roughly 31%.

Figure 5.3 shows the two call sites. A toy language model, `ToyLM`, with a vocabulary of 100 tokens and $d = 16$, shows the mechanics.

Tying is one assignment, and it is aliasing, not copying:

```
class ToyLM(nn.Module):
    def __init__(self, vocab_size, d, tied=True):
        super().__init__()
        self.emb = nn.Embedding(vocab_size, d)
        self.body = nn.Linear(d, d)
        self.head = nn.Linear(d, vocab_size, bias=False)
        if tied:
            self.head.weight = self.emb.weight   # one tensor, two roles

    def forward(self, tokens):
        return self.head(torch.relu(self.body(self.emb(tokens))))

lm = ToyLM(vocab_size=100, d=16)
```

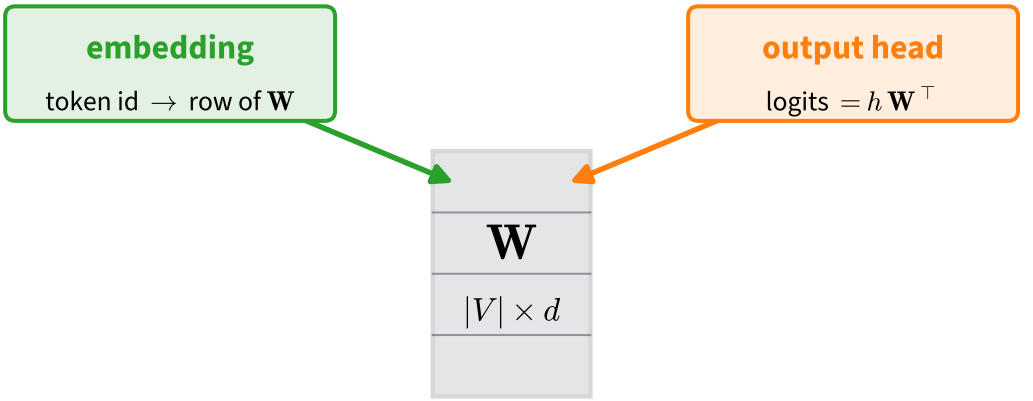


Figure 5.3: One weight matrix W, two call sites: the embedding looks up a row by token id, the output head multiplies by its transpose to produce logits. Tying means both point at the same tensor, so gradients from both uses sum into one place.

```
lm.head.weight is lm.emb.weight
```

```
True
```

Is the shared table counted once or twice? We compare the tied model with an untied twin.

The module system understands the aliasing. The traversal of the parameters reports the shared tensor once, so the optimizer also updates it once per step.

```
untied = ToyLM(vocab_size=100, d=16, tied=False)
(sum(p.numel() for p in lm.parameters()),
 sum(p.numel() for p in untied.parameters()))
```

```
(1872, 3472)
```

The tied model has $1600+272 = 1872$ parameters: the table and the body layer. The untied one adds a second 100×16 table, 3472 in all.

The state dict, by contrast, keeps both names, `emb.weight` and `head.weight`, pointing at the same storage. This is deliberate: a checkpoint saved from a tied model then loads into either a tied or an untied one.

```
sd = lm.state_dict()
(sd['emb.weight'].data_ptr() == sd['head.weight'].data_ptr(),
 [name for name, _ in lm.named_parameters()])
```

```
(True, ['emb.weight', 'body.weight', 'body.bias'])
```

What happens to the gradients? During backpropagation each use of the tensor contributes a gradient, and the contributions add up in the gradient of the one shared tensor. Let $\mathbf{W}_{\text{emb}}$ and $\mathbf{W}_{\text{head}}$ be the two tables of the untied model. If both hold the values of the tied table $\mathbf{W}$, the two models compute the same function, and the chain rule gives

$$\frac{\partial L}{\partial \mathbf{W}} = \frac{\partial L}{\partial \mathbf{W}_{\text{emb}}} + \frac{\partial L}{\partial \mathbf{W}_{\text{head}}}. \tag{5.3}$$

The tied gradient is the sum of the untied gradients. We check this by loading the tied model's values into the untied one and running the same backward pass through both:

```
untied.load_state_dict(lm.state_dict())   # same values, separate tensors
tokens = torch.randint(0, 100, (2, 8))
lm(tokens).sum().backward()
untied(tokens).sum().backward()
torch.allclose(lm.emb.weight.grad,
               untied.emb.weight.grad + untied.head.weight.grad)
```

```
True
```

5.2.3.2 Frozen Parameters

Fine-tuning adapts a model trained on one task to another (Section 32.2). A common recipe keeps the pretrained layers fixed and trains only a new output layer, which needs little data and little memory. Keeping a parameter fixed is called *freezing* it. We freeze everything but the output layer of a fresh copy of our residual MLP, which should leave 650 of its 18,634 parameters trainable.

In PyTorch, setting `requires_grad = False` stops a parameter from accumulating a `.grad`. Backpropagation still passes through operations that involve the frozen tensor, to reach trainable parameters in earlier layers:

```
finetune = nn.Sequential(nn.Linear(20, 64), ResidualBlock(64),
                         ResidualBlock(64), nn.Linear(64, 10))
for p in finetune[:-1].parameters():
    p.requires_grad = False

(sum(p.numel() for p in finetune.parameters() if p.requires_grad),
 sum(p.numel() for p in finetune.parameters()))
```

```
(650, 18634)
```

One Adam step should now move the head and nothing else. We compare every parameter before and after it.

The optimizer receives only the trainable parameters.

```
head_opt = torch.optim.Adam(
    (p for p in finetune.parameters() if p.requires_grad), lr=0.1)
before = [p.clone() for p in finetune.parameters()]
finetune(X).sum().backward()
```

```
head_opt.step()
[not torch.equal(b, p) for b, p in zip(before, finetune.parameters())]
```

```
[False,
 False,
 False,
 False,
 False,
 False,
 False,
 False,
 False,
 False,
 True,
 True]
```

Only the last two entries, the head's weight and bias, changed. Because `head_opt` received only the head, it holds Adam's running averages for 650 parameters.

Freezing has two limits, and neither raises an error. The first is merely wasteful: a freeze does not reclaim optimizer state that already exists. An optimizer that was built over all parameters and has taken a step keeps their running averages, 8 bytes per parameter, whatever is frozen afterwards. For our 18,634 parameters nobody will mind. The last exercise works out what the same oversight means for the 8B model.

The second limit matters more. Freezing stops gradients, and some state changes without one. Batch normalization (Section 7.3) updates its running mean and variance in the *forward pass*, whenever the layer runs in training mode. No optimizer is involved, so there is no reason why freezing the layer's parameters should hold its statistics still. We pass one batch through such a layer in training mode, without any optimizer step, and ask whether the running mean is unchanged.

The running statistics are buffers. We freeze the weight and bias of a `BatchNorm1d` layer:

```
bn = nn.BatchNorm1d(4)
for p in bn.parameters():
    p.requires_grad = False
before = bn.running_mean.clone()
bn(torch.randn(8, 4) + 3)          # training mode: the statistics update
torch.allclose(bn.running_mean, before)
```

```
False
```

The running mean moved. `requires_grad` and `.eval()` are independent switches: the first stops gradients, the second stops the behaviors of training mode, such as updates of running statistics and dropout. To keep a BatchNorm layer fixed during fine-tuning we need both.

5.2.3.3 Weight Averages

A last kind of state is derived from the parameters but not trained: an *exponential moving average* (EMA) of the weights, kept next to the trained ones and used for evaluation. After every optimizer step, the average $\bar{\mathbf{w}}$ moves a fraction of the way toward the current weights $\mathbf{w}$,

$$\bar{\mathbf{w}} \leftarrow 0.9\,\bar{\mathbf{w}} + 0.1\,\mathbf{w}, \tag{5.4}$$

so it lags behind the iterates and smooths out their fluctuations. Whether it evaluates better than the last iterate is a question for Section 7.6, which measures it. All that concerns us here is that the average is state: like BatchNorm statistics, it changes outside backpropagation and belongs in the checkpoint. We start the average at the current weights and continue the fine-tuning above for five steps.

PyTorch provides the mechanism in `torch.optim.swa_utils`: `AveragedModel` keeps a copy of the model and, given the EMA averaging function, blends the live weights into the copy whenever we call `update_parameters`. Its first call copies the weights:

```
ema_model = torch.optim.swa_utils.AveragedModel(
    finetune, multi_avg_fn=torch.optim.swa_utils.get_ema_multi_avg_fn(0.9))
ema_model.update_parameters(finetune)   # the first call copies the weights
for _ in range(5):
    head_opt.zero_grad()
    finetune(X).sum().backward()
    head_opt.step()
    ema_model.update_parameters(finetune)
float((ema_model.module[3].bias - finetune[3].bias).detach().abs().max())
```

```
0.36855900287628174
```

The printed number is the gap between the averaged and the current bias of the head, and it is 0.3686 on every run. This is no accident, and we can predict it. The loss is the sum of the outputs, so the gradient of each bias entry is the batch size, 2, at every step. Adam divides its average of the gradient by the square root of its average of the squared gradient, so a gradient that never changes moves each entry by the learning rate, 0.1, per step. If the weights move by 0.1 per step, the gap e between the average and the weights obeys $e \leftarrow 0.9\,(e + 0.1)$, which starts at 0 and reaches 0.3686 after five steps. The average carries a memory of earlier weights that the weights themselves have lost, so a checkpoint that dropped it could not resume the average where it stopped.

Every parameter we met in this section, whether tied, frozen, or averaged, started from a value that some rule chose. The next section asks which rule to choose, and how to apply it over the tree.

5.2.4 Summary

A model's state is a tree of named tensors, and registration decides which consumers see each one: the optimizer sees the parameters, while checkpoints and device moves also see buffers, persistent tensors such as running statistics that receive no gradient. Unregistered tensors are left behind. Training with Adam in fp32 stores 16 bytes per parameter before any activation: weight, gradient, and Adam's two running averages. For an 8-billion-parameter model that is 128 GB. Tying gives two layers one tensor, which is counted and updated once and receives the sum of the gradients of its uses. Freezing keeps parameters from changing, but it does not free optimizer state that already exists, and it does not stop the updates of running statistics in training mode.

5.2.5 Exercises

1. code **Running statistics.** This problem needs only one fact about the batch normalization layer of Section 7.3, the update below. In training mode the layer normalizes with the minibatch mean $\hat{\mu}_{\mathcal{B}}$ and variance, and each training step updates the running mean $\bar{\mu}$ by

$$\bar{\mu} \leftarrow \rho\,\bar{\mu} + (1-\rho)\,\hat{\mu}_{\mathcal{B}}. \tag{5.5}$$

   1. A fresh layer starts from $\bar{\mu} = \mathbf{0}$. Predict $\bar{\mu}$ after the layer has processed k batches in training mode, each with minibatch mean 3 in every feature. Check the prediction for $k = 1, 5, 20$ on a fresh layer over 4 features, without any optimizer step, and read off the value of ρ the layer uses by default.
   2. Suppose you registered $\bar{\mu}$ as a trainable parameter and removed the update above. What gradient does the training loss send to $\bar{\mu}$? How do the training loss and the evaluation accuracy then compare with those of an unmodified model?
   3. Find a loss in $\bar{\mu}$ alone whose gradient step with learning rate η on each minibatch reproduces the removed update, and state the η that matches ρ.

2. **Parameters and buffers.** Section 5.2.1.2 separates trained parameters from persistent state that receives no gradient. State a one-line rule that classifies every entry of a model's state as one or the other, and check it against the state of `standardizer` in Section 5.2.1.2 and against the batch normalization layer `bn` of Section 5.2.3.2, including any integer counter it keeps.

3. code **Memory footprint.** Write a helper that reports, for each top-level block of `net`, its number of state entries and its bytes under fp32 Adam training with every parameter trained.

   1. Which block dominates? At what output width would the output layer overtake one residual block?
   2. Insert a batch normalization layer over the 64 hidden features between the two residual blocks. Extend the helper so that it costs the layer's running statistics correctly, and report each block's bytes per entry. For which blocks is the share of the total bytes larger than the share of the total entries, and for which is it smaller? Explain both directions.

4. code **Tied parameters.** Copy the tied `lm` with `copy.deepcopy`, as `AveragedModel` does when it clones a model for weight averaging. Check with `is` whether the head and the embedding of the copy share one tensor, and whether that tensor is the original's. Explain both results. Then cast the original with `lm.to(torch.float64)` and check again whether its head and embedding share one tensor.

5. code **Tying and freezing.** Consider the tied `ToyLM` of this section.

   1. Freeze `lm.emb.weight` but leave `lm.head` alone. How many trainable parameters remain, and what does this show about the interaction between tying and freezing?
   2. Starting from the values of `lm`, build a model whose embedding stays fixed while its head adapts, and verify both properties after one optimizer step. How many parameters does it store and how many does it train, compared with a fully trainable `lm`?

6. **Fine-tuning memory.** A language model has 8 billion parameters. Its output layer is a $4096 \times 128{,}256$ matrix, not tied to the embedding. All state is fp32, and the optimizer

is Adam.

1. How many bytes do weights, gradients, and Adam's running averages take when every parameter is trained? How many 80 GB GPUs does that state alone fill?
2. Freeze everything except the output layer, and build the optimizer over the output layer only. How many bytes does the persistent state take now, and does it fit on one 80 GB GPU?
3. Suppose instead that an optimizer built over all parameters took one step before the freeze and is kept afterwards. How many bytes remain allocated, and does the state still fit?

5.3 Initialization

Every network we have trained began from random weights, and we have mostly let the framework draw them. Section 4.4 showed that the draw matters: if each layer scales the signal up or down a little, a deep stack scales it up or down a lot, and the network does not train.

That section also gave the cure for a single layer. Draw each weight with standard deviation $g/\sqrt{n_{\text{in}}}$, where n_{in} is the number of inputs of the layer and the *gain* g depends on what follows it: $g = \sqrt{2}$ before a ReLU, which is He initialization (Section 4.4.2.2), and $g = 1$ when no nonlinearity follows. The layer then passes on the mean square of its input unchanged.

One might hope that a rule for each layer settles the matter. It does not. In a residual network the activations grow even though every layer follows the rule. Each residual block adds the output of a small network, its body f (Section 5.1), to its input. If the body passes on the mean square of its input, and its output is uncorrelated with that input, the sum has twice the mean square of the input. After 32 blocks the standard deviation of the activations has grown by a factor of $2^{16} \approx 65{,}000$, before training has started.

The right initialization therefore depends on how the layers are connected, not only on each layer, and we need a way to apply a chosen rule to chosen parameters of a model. Most of this section is practice, and its first half can be read quickly: what the framework does when we choose nothing, how to apply He initialization to a deep stack of layers, how to write an initializer of our own, and how to edit parameters by hand. The second half is the part to read slowly. There we repair the residual network, by scaling down or zeroing the last layer of each body.

5.3.1 Library Defaults

A framework initializes a layer when we do not, and none of the four frameworks of this book defaults to He initialization (Section 4.4). A statement like this is easy to check, so let us check it. We draw a layer with 512 inputs and 256 outputs and compare its weights with the documented rule. Where the rule is a uniform distribution on $[-b, b]$, the predicted standard deviation is $b/\sqrt{3}$.

PyTorch draws the weight and the bias of an `nn.Linear` from $U(-b, b)$ with $b = 1/\sqrt{n_{\text{in}}}$. (The documentation describes the weight rule as `kaiming_uniform_` with `a=math.sqrt(5)`, a variant of He initialization. The name should not mislead: the algebra reduces to the bound above, and to one sixth of the variance that He initialization prescribes.)

```
layer = nn.Linear(512, 256)
bound = 1 / math.sqrt(layer.in_features)
w = layer.weight
print(f'range [{w.min():.4f}, {w.max():.4f}], predicted bound {bound:.4f}')
print(f'std {w.std():.4f}, predicted {bound / math.sqrt(3):.4f}')
```

```
range [-0.0442, 0.0442], predicted bound 0.0442
std 0.0255, predicted 0.0255
```

The weights follow the documented rule. A network with one or two hidden layers tolerates such a default; a deep one may not. When should we override it? Two reasons concern the scale of the signal: a deep network without normalization layers, where the factor of each layer compounds across depth, and an architecture whose structure the per-layer rule ignores, such as the residual networks of this section. Two more are practical: reproducing a paper whose results depend on its initialization, and parameters we create by hand.

The last case needs particular care in PyTorch: `nn.Parameter(torch.empty(...))` holds whatever bytes the allocator happened to return.

5.3.2 Applying Initializers

The defaults apply one fixed rule to each type of layer. To use He initialization, or the residual scaling later in this section, we have to apply a rule of our own, to a whole model or to selected parameters. Let us start with the whole model: a stack of ten linear layers of width 256, each followed by a ReLU. We leave out the biases, so that the weights alone set the scale, and feed the stack 1,024 inputs with entries drawn from $\mathcal{N}(0, 1)$. We measure the mean square of the output under the default initialization, apply He initialization to every layer, and measure again.

`net.apply(fn)` calls `fn` on every submodule of the tree, children first (Section 5.1.4), and the function decides by the type of each module what, if anything, to do with it. The routines in `torch.nn.init` follow PyTorch's convention that a trailing underscore means *in place*: `nn.init.kaiming_normal_(w)` overwrites `w` rather than returning a new tensor. This matters: were we to replace the tensor that the module registered, an existing optimizer would go on updating a tensor that the model no longer uses. PyTorch names He initialization after its first author, Kaiming He, and `nonlinearity='relu'` selects the gain $\sqrt{2}$. The default has one sixth of He's variance, so each layer with its ReLU multiplies the mean square by $1/6$, and ten layers by $6^{-10} \approx 1.7 \times 10^{-8}$.

```
net = nn.Sequential(*[module for _ in range(10) for module in
                      (nn.Linear(256, 256, bias=False), nn.ReLU())])
X = torch.randn(1024, 256)
print(f'default: mean square of the output {net(X).pow(2).mean():.2g}')

def init_he(module):
    if isinstance(module, nn.Linear):
        nn.init.kaiming_normal_(module.weight, nonlinearity='relu')

net.apply(init_he)
print(f'He:      mean square of the output {net(X).pow(2).mean():.2g}')
```

```
default: mean square of the output 1.7e-08
He:      mean square of the output 0.8
```

Under the default, the mean square of the output is orders of magnitude below that of the input, because each layer multiplies it by the same factor below one. After He initialization it lies within a factor of about two of 1, the mean square of the input. That it is not 1 itself is no fault of the rule. The factor of each layer fluctuates with its random weights, and ten such factors multiply.

This pattern is the whole interface. Later chapters package it as helper functions, `init_cnn` for convolutional networks and `init_seq2seq` for encoder-decoder models, but each is a function like `init_he` handed to `apply`.

5.3.2.1 Truncated Normals

A normal distribution gets the variance right, but its tails are unbounded. That is harmless for one draw. At scale, a value far in the tail is a near-certainty: among the 10^8 weights of a large model, dozens land beyond five standard deviations. Large language and vision models such as BERT and the vision transformer use a *truncated normal* instead, which cuts the tails off at a fixed multiple of the nominal standard deviation (Devlin et al. 2018; Dosovitskiy et al. 2021). Cutting the tails also removes variance, which some initializers restore by rescaling and others do not.

`nn.init.trunc_normal_` takes absolute cutoffs `a` and `b`, which default to ± 2 and so mean ± 2 standard deviations only when `std=1`. With `std=0.02` the defaults would cut at 100 standard deviations, which is to say not at all. A cut at two standard deviations must state the bounds:

```
w = torch.empty(1000, 1000)
nn.init.normal_(w, std=0.02)
print(f'normal:    std {w.std():.4f}, max weight {w.abs().max():.4f}')
nn.init.trunc_normal_(w, std=0.02, a=-0.04, b=0.04)
print(f'truncated: std {w.std():.4f}, max weight {w.abs().max():.4f}')
```

```
normal:    std 0.0200, max weight 0.0979
truncated: std 0.0176, max weight 0.0400
```

A million plain normal draws produce entries near $5\sigma = 0.1$; the truncated version guarantees $|w| \leq 0.04$. Its standard deviation falls from 0.02 to about 0.0176, the variance that the cut removed. Recipes with a fixed standard deviation, such as BERT's, accept this smaller variance; a recipe whose derivation needs a target variance should rescale after truncating.

5.3.3 Custom Initializers and Direct Edits

When no library routine draws what we want, we write the initializer ourselves, and sometimes we set values by hand. Both rest on the interface of an initializer:

In PyTorch, an initializer is a function that overwrites a parameter in place.

5.3.3.1 A Rule with a Gain of Our Choice

Our example is the rule of the introduction with the gain left open: a normal distribution with standard deviation $g/\sqrt{n_{\text{in}}}$ for any g. He initialization is the case $g = \sqrt{2}$. The residual networks at the end of this section need the rule for the last layer of each body with three

more gains, 1, $1/\sqrt{N}$ for a stack of N blocks, and 0, so we write it once, as a function of the gain.

The function follows the convention of `torch.nn.init`: it overwrites the tensor it is given, and its name ends in an underscore. A PyTorch weight has shape $(n_{\text{out}}, n_{\text{in}})$:

```
def scaled_normal_(weight, gain):
    nn.init.normal_(weight, std=gain / math.sqrt(weight.shape[1]))
```

5.3.3.2 Direct Edits

An initializer sets every entry of a parameter by one rule. We can also change values by hand, after initialization or after training. Halving the weights of the last layer of the stack above halves its outputs before the ReLU, and the ReLU passes the factor on, so the mean square of the output should fall by a factor of four.

The edit needs `torch.no_grad()`: parameters are leaf tensors that track gradients, and PyTorch rejects in-place arithmetic on them unless we declare that the change is not part of any computation to differentiate. (The `nn.init` routines run under their own `no_grad` internally, which is why `init_he` needed none.)

```
before = net(X).pow(2).mean()
with torch.no_grad():
    net[-2].weight *= 0.5
print(f'{before:.2g} before, {net(X).pow(2).mean():.2g} after')
```

```
0.8 before, 0.2 after
```

The second number is a quarter of the first.

5.3.4 Initialization of Residual Networks

5.3.4.1 Variance Growth across Blocks

The introduction claimed that a residual network explodes although every layer follows the rule. Let us do the calculation. A residual block computes $\mathbf{x}_{k+1} = \mathbf{x}_k + f_k(\mathbf{x}_k)$, where the body f_k is a small network. We use the body of the blocks of Section 5.1, widened fourfold as in many residual networks: a linear layer from d to $4d$ features, a ReLU, and a linear layer from $4d$ back to d, the *output projection* of the body. The first layer gets He initialization, gain $\sqrt{2}$, because a ReLU follows it. The output projection gets standard deviation $g/\sqrt{4d}$ with a gain g that we leave open for now; $g = 1$ is the choice of Section 4.4 for a layer that feeds no nonlinearity. All biases start at zero.

How does the mean square of the activations change across one block? Write m_k for the mean square of the entries of $\mathbf{x}_k$. The first layer doubles it and the ReLU halves it again (Section 4.4.2.2), so together they pass on m_k. The output projection multiplies it by $4d \cdot g^2/(4d) = g^2$, so the output of the body has mean square $g^2 m_k$. So far this is routine. The step that needs an argument is the sum. The output of the body is uncorrelated with $\mathbf{x}_k$, because the weights of the output projection have mean zero and are independent of $\mathbf{x}_k$. The mean square of the sum is therefore the sum of the mean squares,

$$m_{k+1} = m_k + g^2 m_k = (1 + g^2)\, m_k. \tag{5.6}$$

Each block multiplies the mean square by $1 + g^2$, a factor above one for every $g \neq 0$. With $g = 1$ it doubles, and after N blocks the standard deviation has grown by $2^{N/2}$: a factor of 2 after 2 blocks, 16 after 8, and $65{,}536$ after 32. This resolves the puzzle of the introduction. Every layer was initialized by the rule for one layer, and the network as a whole explodes all the same, because that rule knows nothing of the skip connection, which adds the input back in every block.

5.3.4.2 Scaling Down the Output Projection

The factor $1 + g^2$ per block compounds to $(1 + g^2)^N$ after N blocks. To keep the product bounded as the network gets deeper, the gain must shrink with N. How fast? The choice $g^2 = 1/N$, that is, scaling the output projection by $1/\sqrt{N}$, turns out to be just enough:

$$m_N = \left(1 + \frac{1}{N}\right)^N m_0 \leq e\, m_0, \tag{5.7}$$

since $(1 + 1/N)^N$ increases toward e. The standard deviation then grows by at most $\sqrt{e} \approx 1.65$, however deep the network: by 1.5 after 2 blocks, 1.60 after 8, and 1.64 after 32. The rule is in practical use. The language model GPT-2 (Radford et al. 2019) scales the output projection of every residual body (often called a residual *branch*) by $1/\sqrt{N}$, where N counts the bodies.

5.3.4.3 Zero Output Projections

The extreme choice is $g = 0$. At first sight this looks reckless: zero-initializing *every* weight is fatal, since all units of a layer then stay identical forever (Section 4.4). Zero-initializing only the output projection is a different matter. The body then contributes nothing, each block is the identity map, and the network starts as a shallow function whose depth comes into play as training proceeds. Symmetry is not a problem, because the body's first layer keeps its random weights. On the first backward pass, the output projection receives a nonzero gradient, because its input is nonzero, while the first layer of the body receives zero gradient through the zero projection. Once the projection has moved off zero, gradient reaches the whole body.

5.3.4.4 Measuring Variance across Depth

Three predictions as definite as these ought to be checked by measurement. `WideResidualBlock` is the residual block of Section 5.1 with its body widened fourfold and an explicit initialization: the gain of the output projection is a constructor argument, and both layers draw their weights from the scaled normal initializer above, the first with gain $\sqrt{2}$ and the output projection with that gain:

```
class WideResidualBlock(nn.Module):
    def __init__(self, d, out_gain):
        super().__init__()
        self.body = nn.Sequential(nn.Linear(d, 4 * d), nn.ReLU(),
```

```
                                nn.Linear(4 * d, d))
        for layer, gain in ((self.body[0], math.sqrt(2)),
                            (self.body[2], out_gain)):
            scaled_normal_(layer.weight, gain)
            nn.init.zeros_(layer.bias)

    def forward(self, X):
        return X + self.body(X)
```

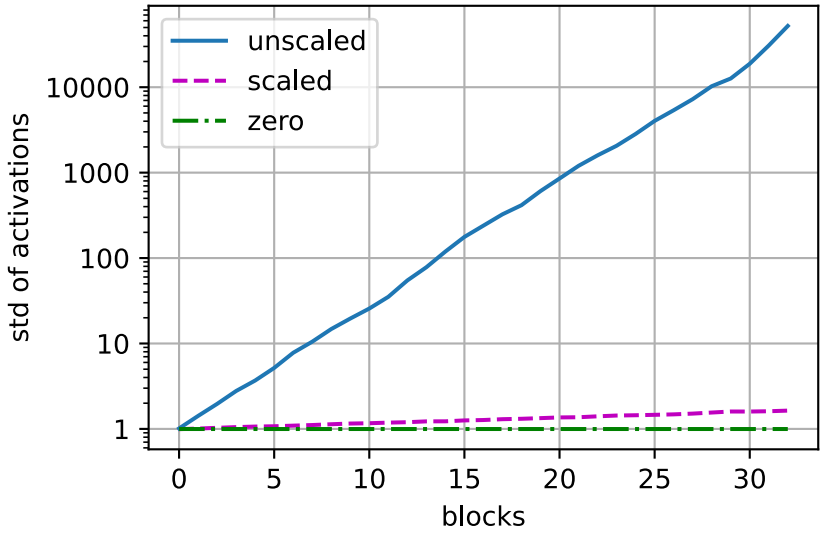
The function below stacks `num_blocks` blocks of width 64, feeds them 256 examples with entries drawn from $\mathcal{N}(0, 1)$, and records the standard deviation of the activations at the input and after every block:

```
def std_across_depth(num_blocks, out_gain):
    net = nn.Sequential(*[WideResidualBlock(64, out_gain)
                          for _ in range(num_blocks)])
    X = torch.randn(256, 64)
    stds = [X.std().item()]
    with torch.no_grad():
        for block in net:
            X = block(X)
            stds.append(X.std().item())
    return stds
```

The three choices of gain, unscaled, scaled, and zero, set the gain of the output projection to 1, to $1/\sqrt{N}$, and to 0. We print the standard deviation after the last block for $N = 2, 8, 32$ and plot it after every block for $N = 32$:

```
gains = {'unscaled': lambda n: 1.0, 'scaled': lambda n: n ** -0.5,
         'zero': lambda n: 0.0}
profiles = {n: [std_across_depth(n, gain(n)) for gain in gains.values()]
            for n in (2, 8, 32)}
print(f'{"N":>3}' + ''.join(f'{name:>10}' for name in gains))
for n, stds in profiles.items():
    print(f'{n:>3}' + ''.join(f'{s[-1]:>10.3g}' for s in stds))
d2l.plot(list(range(33)), profiles[32], 'blocks', 'std of activations',
         legend=list(gains), yscale='log', figsize=(4.5, 3))
```

```
  N  unscaled    scaled      zero
  2      2.01      1.47     0.998
  8      13.9      1.61         1
 32  5.21e+04      1.64     0.997
```



The measurements follow the predictions. The unscaled column grows by about $\sqrt{2}$ per block, a straight line on the logarithmic axis. After 32 blocks it lands between about 20,000 and 200,000, in two runs out of three below the predicted 65,536. The spread is no fault of the measurement. The prediction is the square root of the expected mean square; the factor of each block fluctuates with its random weights, the fluctuations compound, and the typical run lies below the square root of the average. We have not trained this stack, and with initial activations in the tens of thousands there is little reason to try. The scaled column stays between about 1.4 and 1.9 at every depth, near the predicted 1.5 to 1.65, and the zero column keeps the standard deviation of the input, 1, since every block is the identity.

A forward pass of this kind catches an unusable initialization before training begins. It cannot rank the two usable ones. Both keep the activations at scale, and they differ only in how training starts: with every body active under the $1/\sqrt{N}$ scaling, and with none under the zero projection.

Initialization only sets the starting point. Once training moves the weights, nothing in this section keeps the activations at scale. That is the task of the normalization layers of Section 7.3.

5.3.5 Summary

A framework initializes its layers by default, with a fixed rule that is not He initialization, so a deep model without normalization layers needs an explicit rule. Initializers are applied over the tree of a model: as constructor arguments, as a function applied to every submodule, or by assigning new values to selected parameters. Truncated normals bound the largest weights, and an initializer of our own takes a few lines of code. The per-layer rules of Section 4.4 are not enough for a residual network: each block adds its body to its input and multiplies the mean square of the activations by $1 + g^2$, where g is the gain of the body's output projection. With $g = 1$ the standard deviation grows by $\sqrt{2}$ per block; scaling the output projection by $1/\sqrt{N}$ bounds the growth by $\sqrt{e}$ at any depth N, and a zero output projection makes every block start as the identity.

5.3.6 Exercises

1. **Truncated normal.** For a normal distribution truncated at $\pm 2\sigma$: what fraction of draws does the cut discard, and by how much does it shrink the standard deviation? Neither quantity has an elementary closed form; express both in terms of the standard normal density and distribution function and evaluate them using tabulated values. Verify the standard deviation against the printed output of the demonstration in Section 5.3.2.1, and the discarded fraction by counting the entries of its plain normal draw that lie beyond $\pm 2\sigma$.

2. **Zero initialization.** Zero-initialize *both* layers of every body, weights and biases, instead of just the output projection. The forward pass still returns the input. Which parameters receive a nonzero gradient on the first backward pass, and on any later one? Is this the same failure as in the symmetry argument of Section 4.4?

3. code **Normalized inputs.** Many residual networks normalize the input of each body: the body reads $\mathbf{x}_k / r_k$, where r_k is the root mean square of the entries of $\mathbf{x}_k$ for each example, instead of $\mathbf{x}_k$.

    1. Show that the body output then has mean square g^2, whatever the scale of $\mathbf{x}_k$, and that with $m_0 = 1$ the mean square after N blocks is $1 + Ng^2$. How does the growth with depth compare with the unnormalized blocks of this section?
    2. Which gain keeps the mean square at the end of the stack equal to 2 for every N? Compare it with the scaling of GPT-2.
    3. Add the normalization to `WideResidualBlock`, measure the standard deviation across depth for $g = 1$ and $g = 1/\sqrt{N}$ at $N = 32$, and compare with your predictions and with the measured plot in Section 5.3.4.4.

4. code **Kaiming initialization.** Implement the Kaiming (He) initializers of Section 4.4.2.2, uniform and normal, with an explicit `nonlinearity` argument that switches the gain between `'relu'` and `'linear'`.

    1. Check both on the weight of a layer with 512 inputs and 256 outputs by mean, standard deviation, and range: the normal initializer against the scaled normal initializer of this section, and the uniform initializer against its predicted bound and standard deviation. Explain any difference in range.
    2. Apply your normal initializer to the stack of Section 5.3.4.4 with the `'relu'` gain on both layers of every block. Predict the factor by which each block multiplies the mean square, and check it against a measured curve added to the plot of Section 5.3.4.4.

    Adapted from CMU 10-714, Homework 2 and Homework 4.

5. code **Gradients across depth.** ⊠ The backward pass through a block multiplies the gradient by the transposed Jacobian of $\mathbf{x} \mapsto \mathbf{x} + f(\mathbf{x})$.

    1. Repeat the argument of this section for the backward pass. Show that the body multiplies the mean square of the gradient by g^2, although the backward pass sums over the outputs of each layer rather than its inputs, so that each block again multiplies it by $1 + g^2$.
    2. Verify the prediction for the three choices of gain at $N = 32$: compute the gradient of the sum of the outputs of the stack with respect to its input, and compare its mean square with that of the gradient at the output.

6. code **Data-driven calibration.** ▯ Initialize the stack of Section 5.3.4.4 with the unscaled choice of gain, then calibrate it block by block on one minibatch of unit-variance inputs. For each block in turn, run the minibatch forward to that block, measure the standard deviation of its body output $f_k(\mathbf{x}_k)$, and rescale the block's output projection so that this standard deviation becomes $1/\sqrt{N}$. This adapts the layer-sequential unit-variance initialization of Mishkin and Matas (2016) to a residual stack.
    1. Explain why the blocks must be calibrated in order rather than all at once from a single forward pass.
    2. Explain why the calibration targets the body output rather than the block output $\mathbf{x}_k + f_k(\mathbf{x}_k)$.
    3. Predict the standard deviation at the end of the calibrated stack. Plot the calibrated depth-versus-standard-deviation curve next to that of the scaled choice of gain, and explain the difference.
7. code **Default biases.** The stack of ten ReLU layers in this section left out the biases, and under PyTorch's default weights its mean square fell by a factor of 6 per layer. Restore the biases, which PyTorch draws from the same $U(-1/\sqrt{n_{\text{in}}}, 1/\sqrt{n_{\text{in}}})$ as the weights, and print the mean square after every layer. Explain why it stops shrinking after a few layers, and predict the value at which it levels off.

5.4 Custom Layers and Functions

A framework's library holds the layers that many people needed yesterday: fully connected layers, activation functions, dropout. Research moves faster than libraries, and a layer that a paper proposes can be missing from them for years. RMSNorm, a layer that rescales each activation vector to a fixed size, is an example. It was published in 2019 (Zhang and Sennrich 2019), and PyTorch added it to its library of layers in 2024. Until then, each PyTorch codebase that used it carried its own implementation or borrowed one from another package. How do we write such a layer ourselves, so that it behaves like a built-in one? Its parameters must reach the optimizer, its state must be saved and moved to the GPU, and it must fit into containers next to other layers.

Sometimes the trouble lies with the derivative instead. To fit a large model into a small memory, its weights are often stored as 4-bit integers: the weights of the 8B model of Section 5.1 then take 4 GB, a quarter of what 16-bit floating-point numbers need. Storing integers means rounding, and rounding has derivative zero almost everywhere, so a weight that is rounded in the forward pass receives no gradient and never learns.

This section writes both kinds of extension. A custom layer is a subclass of the module class of Section 5.1. Everything the framework does to a whole model, from collecting parameters for the optimizer to saving a checkpoint, walks the registered state, so a layer that registers its state needs no further code to take part. The ideas here are few. Most of the section shows the form in which each framework expects them. We start with a layer without state, then write RMSNorm with a learned parameter, then a layer with fixed state that must be saved but never trained. Finally we replace the derivative of an operation with a rule of our own, which lets gradients pass through rounding.

5.4.1 Layers without Parameters

Let us start with the smallest custom layer, one with no state at all. `CenteredLayer` subtracts from its input the mean over all of its entries, the batch axis included (an exercise asks what that implies). It inherits from the module class and supplies only the forward method; there is nothing to set up.

```
class CenteredLayer(nn.Module):
    def __init__(self):
        super().__init__()

    def forward(self, X):
        return X - X.mean()
```

The inputs 1 to 5 have mean 3, so the output should run from -2 to 2:

```
layer = CenteredLayer()
layer(torch.tensor([1.0, 2, 3, 4, 5]))
```

```
tensor([-2., -1.,  0.,  1.,  2.])
```

The class has the same interface as a built-in layer, so a sequential container can call it like any other child. Placed after a linear layer, it centers the linear layer's outputs. Their mean should be zero. In floating-point arithmetic it comes out as exactly zero only by luck; a number of size 10^{-8} or smaller is rounding error and no cause for alarm.

```
net = nn.Sequential(nn.Linear(8, 128), CenteredLayer())
Y = net(torch.rand(4, 8))
Y.mean()
```

```
tensor(2.0955e-09, grad_fn=<MeanBackward0>)
```

5.4.2 Layers with Parameters

In Section 4.4 we saw that the scale of the activations can grow or shrink from one layer to the next, by a factor that depends on the weights. A *normalization layer* removes this dependence: it rescales each input vector to a fixed size and lets a learned parameter set the size instead. Such layers form a family that we study from Section 7.3 onwards. Here we implement its simplest member, *RMSNorm*.

5.4.2.1 RMSNorm

The root mean square of a vector $\mathbf{x} \in \mathbb{R}^d$ is $\sqrt{\frac{1}{d}\sum_{i=1}^{d} x_i^2}$. RMSNorm divides each example's vector by it and multiplies the result elementwise by a learned *gain* $\mathbf{g} \in \mathbb{R}^d$:

$$\mathrm{RMSNorm}(\mathbf{x}) = \frac{\mathbf{x}}{\sqrt{\frac{1}{d}\sum_{i=1}^{d} x_i^2 + \epsilon}} \odot \mathbf{g}.$$

The small constant ϵ, such as 10^{-6}, keeps the denominator away from zero. With the gain at its initial value $\mathbf{g} = \mathbf{1}$, the output has root mean square close to 1, whatever the scale of the input. For $\mathbf{x} = (3, 4)$ the mean square is $(9 + 16)/2 = 12.5$, the root mean square is 3.536, and the output is $(0.849, 1.131)$, whose mean square is 1. The input $(300, 400)$ gives the same output. Figure 5.4 shows the geometry: RMSNorm keeps the direction of $\mathbf{x}$, discards its length by moving it onto the sphere of radius $\sqrt{d}$, and the gain then stretches each coordinate.

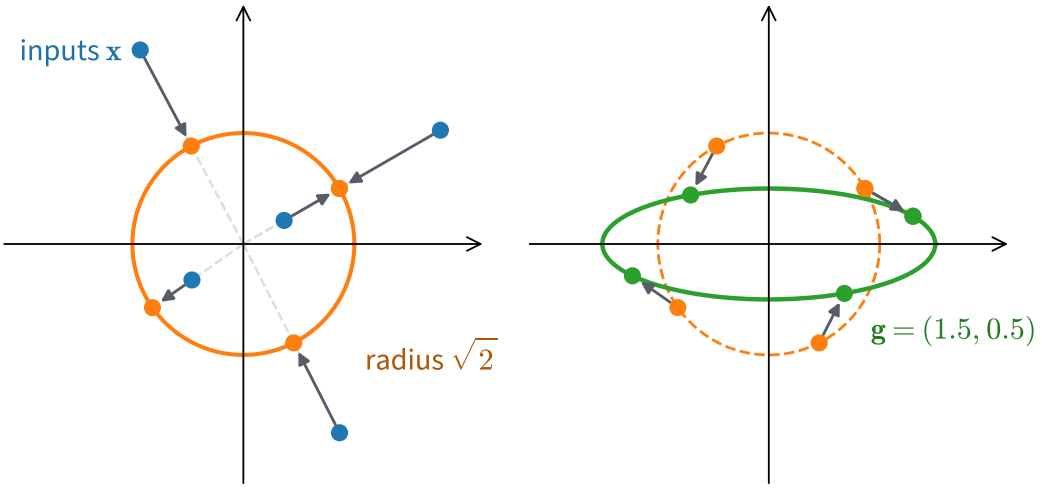


Figure 5.4: RMSNorm in two dimensions. Left: with gain 1, every input (blue) moves along its ray from the origin onto the circle of radius $\sqrt{2}$ (orange); two inputs on the same ray, at lengths 0.6 and 2.9, land on the same point. Right: the gain $\mathbf{g} = (1.5, 0.5)$ stretches the circle into an ellipse with semi-axes $1.5\sqrt{2}$ and $0.5\sqrt{2}$.

RMSNorm simplifies *layer normalization* (Ba et al. 2016), which first subtracts the mean of the entries of each example's vector, as a per-example `CenteredLayer` would, and then adds a learned shift after the gain. Zhang and Sennrich found that the centering contributes little to training, and many recent language models use RMSNorm, among them Llama 3 8B (Grattafiori et al. 2024), the 8B model of Section 5.1.

The gain is to be learned, so the layer must register it as a parameter (Section 5.2). Apart from that, the code below follows the formula.

In PyTorch, assigning a tensor wrapped in `nn.Parameter` to an attribute registers it.

```
class RMSNorm(nn.Module):
    def __init__(self, d, eps=1e-6):
        super().__init__()
        self.gain = nn.Parameter(torch.ones(d))
        self.eps = eps

    def forward(self, X):
        rms = X.pow(2).mean(-1, keepdim=True).add(self.eps).sqrt()
        return self.gain * X / rms
```

We try the layer on the worked example and on the same vector scaled by 100. Both rows should read $(0.849, 1.131)$:

```
norm = RMSNorm(2)
norm(torch.tensor([[3.0, 4.0], [300.0, 400.0]]))
```

```
tensor([[0.8485, 1.1314],
        [0.8485, 1.1314]], grad_fn=<DivBackward0>)
```

5.4.2.2 Registration

Why did we write RMSNorm as a module and not as a function with a gain tensor kept beside it? Because registration makes a custom layer behave like a built-in one in four ways: its parameters are found, it works inside containers, its state is saved and loaded, and it moves between devices. None of this is new. Section 5.1 and Section 5.2 showed all four for the library's layers, so one check on a layer of our own suffices. We place RMSNorm between two linear layers, take one optimizer step, which moves the gain away from its initial ones, and copy the state of the model into a second model with the same structure and its own random initial weights. Afterwards the two compute the same outputs (Section 5.6 writes such state to disk).

An optimizer given `net.parameters()` finds the gain and updates it. `state_dict` returns every registered tensor, the gain included, keyed by its path in the module tree, and `load_state_dict` copies them into the clone. `.to(device)` would move the gain with the rest of the model.

```
net = nn.Sequential(nn.Linear(20, 8), RMSNorm(8), nn.Linear(8, 2))
X = torch.randn(4, 20)
optimizer = torch.optim.SGD(net.parameters(), lr=0.1)
net(X).pow(2).sum().backward()
optimizer.step()
clone = nn.Sequential(nn.Linear(20, 8), RMSNorm(8), nn.Linear(8, 2))
clone.load_state_dict(net.state_dict())
net[1].gain.data, torch.equal(net(X), clone(X))
```

```
(tensor([0.8196, 0.8672, 0.9294, 0.9737, 0.9329, 0.7735, 0.9659, 1.0062]),
 True)
```

The gain is no longer all ones, and the clone computes exactly the outputs of the trained model. None of this required any code in `RMSNorm` beyond the registration of its gain.

5.4.2.3 Comparison with Framework Implementations

The checks so far say that the layer is a good citizen of the framework. They do not say that it computes the right thing. Where the library provides the same layer, we can find out by comparing the outputs of the two implementations.

PyTorch's `nn.RMSNorm` calls its gain `weight`. We give both implementations the same gain, other than all ones, and compare their outputs.

```
ours, native = RMSNorm(8), nn.RMSNorm(8, eps=1e-6)
with torch.no_grad():
```

```
    ours.gain.copy_(torch.linspace(0.5, 1.5, 8))
    native.weight.copy_(ours.gain)
X = torch.randn(16, 8)
torch.allclose(ours(X), native(X))
```

```
True
```

The two agree to floating-point precision. Having written the layer, should we now use it? Not when the library's layer computes what we need: that one is maintained with the library and may run faster, for example by combining several operations into one. A custom layer is the right choice for a new layer, a modification of an existing one, or a layer the library lacks, and then its values, gradients, and saved state need tests like the ones above.

5.4.3 Layers with Fixed State

Some layers hold arrays that are neither learned nor recomputed: they are fixed when the layer is built and must stay the same for the life of the model. An example is a layer of *random features* (Rahimi and Recht 2007). It multiplies a minibatch $\mathbf{X}$ by a random matrix $\mathbf{W} \in \mathbb{R}^{d \times D}$ with standard normal entries, adds a random offset $\mathbf{b} \in \mathbb{R}^{D}$ with entries uniform on $[0, 2\pi]$, and applies the cosine elementwise:

$$\mathbf{Z} = \sqrt{2/D}\, \cos(\mathbf{X}\mathbf{W} + \mathbf{b}). \tag{5.8}$$

Each row of $\mathbf{Z}$ holds D nonlinear features of the corresponding example. Surprisingly, these features need no training: a linear layer on top of them can fit nonlinear functions of the input while only the linear layer is trained.

What matters for us is the state. $\mathbf{W}$ and $\mathbf{b}$ are drawn once and must then not change, so they are not parameters. Yet they must be saved with the model, since the linear layer on top is trained for these features and no others. Arrays of this kind are the *buffers* of Section 5.2.1.2, saved and moved with the model but never trained.

In PyTorch, `register_buffer` registers a tensor as a buffer.

```
class RandomFeatures(nn.Module):
    def __init__(self, d, num_features):
        super().__init__()
        self.register_buffer('W', torch.randn(d, num_features))
        self.register_buffer('b', 2 * torch.pi * torch.rand(num_features))

    def forward(self, X):
        return (2 / self.b.numel()) ** 0.5 * torch.cos(X @ self.W + self.b)
```

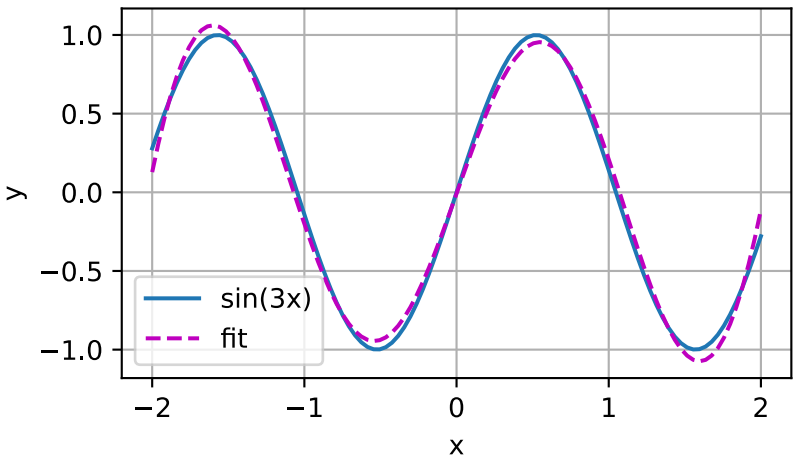
Let us see the features at work. We fit $y = \sin(3x)$ at 100 points evenly spaced in $[-2, 2]$ with a model of two layers, 500 random features followed by a linear layer, and train it by Adam on the mean squared error. The optimizer receives the parameters of the whole model, and the buffers are not among them, so only the linear layer learns:

```
x = torch.linspace(-2, 2, 100)
X, y = x.reshape(-1, 1), torch.sin(3 * x)
```

```
def mse(net, X, y):
    return ((net(X)[:, 0] - y) ** 2).mean()

net = nn.Sequential(RandomFeatures(1, 500), nn.Linear(500, 1))
optimizer = torch.optim.Adam(net.parameters(), lr=0.05)
for _ in range(500):
    optimizer.zero_grad()
    mse(net, X, y).backward()
    optimizer.step()
print(f'mean squared error {mse(net, X, y):.2g}')
d2l.plot(x, [y, net(X)[:, 0].detach()], 'x', 'y',
         legend=['sin(3x)', 'fit'], figsize=(4.5, 2.5))
```

```
mean squared error 0.0032
```



The fit follows the sine closely. Its mean squared error ends below 0.01, against 0.5 for predicting zero everywhere, the mean square of $\sin(3x)$ over the interval.

Now suppose that the checkpoint held only the trained linear layer. A model rebuilt from it draws a new $\mathbf{W}$ and a new $\mathbf{b}$, and the linear layer receives features it was never trained on. We load the linear layer alone into a fresh model, and then the full state, buffers included:

```
fresh = nn.Sequential(RandomFeatures(1, 500), nn.Linear(500, 1))
fresh[1].load_state_dict(net[1].state_dict())
print(f'linear layer only: mean squared error {mse(fresh, X, y):.2g}')
fresh.load_state_dict(net.state_dict())
print(f'full state:        mean squared error {mse(fresh, X, y):.2g}')
```

```
linear layer only: mean squared error 0.94
full state:        mean squared error 0.0032
```

With new random features the error of the trained linear layer rises a hundredfold or more, to the level of predicting zero or worse. With the buffers loaded, it is again the error of the trained model.

`net.state_dict()` holds `0.W` and `0.b` next to the weight and bias of the linear layer. Plain tensor attributes would be missing from it.

The same mechanism holds the fixed tables and running averages of layers in later chapters, such as the running statistics of batch normalization in Section 7.3.

5.4.4 Functions with Custom Gradients

So far we wrote forward computations and let automatic differentiation derive the backward pass. It differentiates the operations we ran, faithfully, and occasionally that is the problem. Recall the model that is trained to work with weights stored as 4-bit integers. It rounds its weights in the forward pass. Rounding is constant between integers, so its derivative is zero almost everywhere, and any weight behind a rounding operation receives a zero gradient:

```
w = torch.tensor([0.9, 1.4, 2.6], requires_grad=True)
w.round().sum().backward()
w.grad
```

```
tensor([0., 0., 0.])
```

The gradient is zero, as calculus says, and useless for training. The *straight-through estimator* (Bengio, Léonard, et al. 2013) replaces it: the forward pass still rounds, and the backward pass treats the rounding as if it were the identity, whose derivative is 1. This is deliberately not the true derivative. Its justification is that the staircase never strays more than $1/2$ from the identity, as Figure 5.5 shows. Automatic differentiation cannot produce such a rule, because it is not the derivative of what the forward pass computes, so we supply the backward pass ourselves.

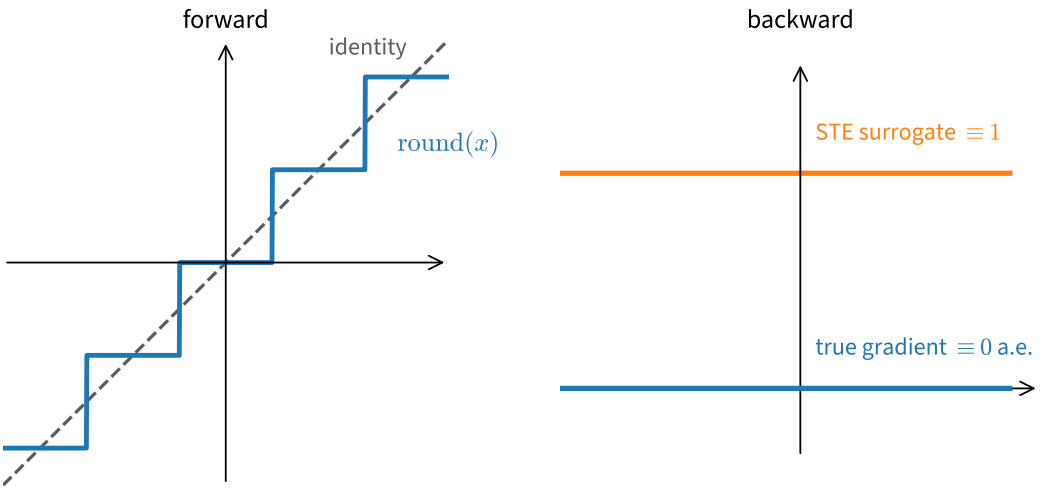


Figure 5.5: The straight-through estimator. Forward: the staircase round(x) stays close to the identity (dashed). Backward: the true derivative of rounding is zero almost everywhere; the estimator uses the derivative of the identity, 1, instead.

In PyTorch the tool is a subclass of `torch.autograd.Function` with the two directions as static methods:

```
class RoundSTE(torch.autograd.Function):
    @staticmethod
    def forward(ctx, X):
        return X.round()

    @staticmethod
    def backward(ctx, grad_output):
        # Treat the forward pass as the identity: pass the gradient on
        return grad_output
```

Two rules govern the class. First, we call it as `RoundSTE.apply(X)`, never through `forward` directly. `apply` inserts the operation into the graph that autograd records. A direct call runs `X.round()` as an ordinary operation, which autograd records with its true derivative, zero; the custom backward pass never runs, and no error says so. Second, `backward` receives the gradient of the loss with respect to the output and returns the gradient with respect to each input. Ours ignores the input values; a backward pass that needs them saves them in `forward` with `ctx.save_for_backward` and reads them from `ctx`.

To check the rule, we multiply the rounded values by known weights $(1, 2, 3)$ and sum. The straight-through estimator should deliver these weights, untouched, as the gradient at $\mathbf{w}$:

```
w.grad = None
y = (RoundSTE.apply(w) * torch.tensor([1.0, 2.0, 3.0])).sum()
y.backward()
w.grad
```

```
tensor([1., 2., 3.])
```

The gradient arrives at $\mathbf{w}$ unchanged, as if the rounding were the identity, while the forward pass computed with the rounded values. Does a weight then learn through the rounding? We fit a linear model whose eight weights are rounded to integers in the forward pass to data generated by weights drawn uniformly from $[-7, 7]$, the range of 4-bit integers. Gradient descent starts from $\mathbf{w} = \mathbf{0}$, once with plain rounding and once with the straight-through estimator. For comparison we also print the loss at the start and the loss of the generating weights rounded to the nearest integers:

```
X = torch.randn(256, 8)
w_true = torch.empty(8).uniform_(-7, 7)
y = X @ w_true

def fit(rnd, X, y, num_steps=100, lr=0.05):
    w = torch.zeros(8, requires_grad=True)
    optimizer = torch.optim.SGD([w], lr=lr)
    for _ in range(num_steps):
        loss = ((X @ rnd(w) - y) ** 2).mean()
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()
    return loss.item()

print(f'start {(y ** 2).mean():.3g}, '
```

```
        f'rounding {fit(torch.round, X, y):.3g}, '
        f'straight-through {fit(RoundSTE.apply, X, y):.3g}, '
        f'rounded target {((X @ w_true.round() - y) ** 2).mean():.3g}')
```

```
start 109, rounding 109, straight-through 1.76, rounded target 0.357
```

With plain rounding the loss never leaves its starting value, since every gradient is zero. With the straight-through estimator it falls from tens or hundreds to about 1, within a small factor of the loss of the rounded generating weights, which is the error that rounding them would cause. A gradient that is deliberately wrong has trained the model.

The identity is one choice of surrogate among several. A quantizer to 4-bit integers also clips its input to the representable range, -8 to 7, and a common variant of the estimator passes the gradient only inside that range and sets it to zero outside, where the clipped output does not change (an exercise implements such a rule). The surrogate is a modeling choice, and the right one depends on the quantizer it stands in for.

For the identity there is a shortcut without a hand-written backward pass. The papers that introduced the estimator describe it in words; this shortcut is the form one usually meets in code. A *stop-gradient* operation is the identity in the forward pass and has derivative zero in the backward pass, so $X + \text{stop}(\text{round}(X) - X)$ computes $\text{round}(X)$ in the forward pass while the backward pass sees only the leading X. The general mechanism is still needed when the surrogate is not the identity, as in the clipped variant. The two definitions agree in value and in gradient:

In PyTorch the stop-gradient operation is the method `detach()`, which returns a tensor with the same values that autograd treats as a constant.

```
def round_ste2(X):
    return X + (X.round() - X).detach()

def grad_of(f):
    w.grad = None
    (f(w) * torch.tensor([1.0, 2.0, 3.0])).sum().backward()
    return w.grad

(torch.allclose(RoundSTE.apply(w), round_ste2(w)),
 torch.allclose(grad_of(RoundSTE.apply), grad_of(round_ste2)))
```

```
(True, True)
```

A custom gradient changes the derivative of a computation built from existing tensor operations; it does not create new operations. Writing new low-level operations for the GPU is a separate craft that this book does not cover. Chapter 13 shows how to get performance out of the operations that the libraries already provide.

What we have not asked of any of these layers is whether their arithmetic is accurate. RMSNorm squares its inputs, and the sum of squares can overflow in low precision. Section 5.5 takes up the number formats themselves.

5.4.5 Summary

A custom layer is a subclass of the module class with a forward method. Registration does the rest: a learned parameter registered with the module is found by the optimizer, saved, loaded, and moved like the parameters of built-in layers, as we checked for the gain of RMSNorm. State that is fixed, such as the random matrix of a random-feature layer, is registered as a buffer, which is saved and moved but never trained. When the derivative of an operation is useless, as for rounding, a custom backward rule such as the straight-through estimator replaces it. A library implementation is preferable when it computes the same thing.

In PyTorch, `nn.Parameter` holds learned state, `register_buffer` registers buffers, and `torch.autograd.Function` supplies a backward rule, invoked through `apply`; `detach()` covers the straight-through case in one expression.

5.4.6 Exercises

1. **Centering over the batch.** `CenteredLayer` subtracts the mean over every entry of its input, the batch axis included. Consider the network of this section, a linear layer with bias **b** followed by `CenteredLayer`.
    1. Show that the output for one example depends on the other examples in the minibatch. State when evaluating the network on one example at a time gives the same outputs as evaluating it on the whole minibatch at once.
    2. Show that adding the same constant to every entry of **b** leaves the output unchanged, and conclude which direction in the space of biases receives zero gradient.
    3. A per-example variant subtracts from each row its own mean over the feature axis, as layer normalization does. State which of the two properties above this variant keeps.
2. code **Scale invariance of RMSNorm.** ⊠ Set $\epsilon = 0$.
    1. Show that $\mathrm{RMSNorm}(c\mathbf{x}) = \mathrm{RMSNorm}(\mathbf{x})$ for every $c > 0$.
    2. Differentiate both sides with respect to c at $c = 1$ and conclude that the Jacobian $\mathbf{J}$ of RMSNorm at $\mathbf{x}$ satisfies $\mathbf{J}\mathbf{x} = \mathbf{0}$.
    3. Conclude that the gradient of any loss with respect to the input $\mathbf{x}$ of RMSNorm is orthogonal to $\mathbf{x}$, and check this with automatic differentiation for a random $\mathbf{x}$ and a random linear loss. What does a small gradient step do to the length of $\mathbf{x}$?
3. code **RMSNorm bias.** Add an optional learned bias **b** to `RMSNorm` (a shift applied after the gain, restoring part of what layer normalization had), and verify that the state dict of a model containing it grows by the expected entry. Then place the layer directly before a linear layer with weight **W** and bias **c**, and show that **b** can be folded into **c** without changing the outputs. What is the folded bias?
4. code **Random features approximate a kernel.** Let $\mathbf{w}$ have independent standard normal entries and b be uniform on $[0, 2\pi]$.
    1. Using $\cos\alpha\cos\beta = \frac{1}{2}(\cos(\alpha-\beta) + \cos(\alpha+\beta))$ and $\mathbb{E}[\cos(\mathbf{w}^\top\mathbf{u})] = \exp(-\|\mathbf{u}\|^2/2)$, show that the features $\mathbf{z}(\mathbf{x})$ of `RandomFeatures` satisfy $\mathbb{E}[\mathbf{z}(\mathbf{x})^\top\mathbf{z}(\mathbf{x}')] = \exp(-\|\mathbf{x}-\mathbf{x}'\|^2/2)$ for every number D of features.

2. Check this numerically for two inputs at distance 1, with $D = 10$, 100, 1,000, and 10,000, averaging the error over several layers for each D. How does the error shrink as D grows?
3. Repeat the fit of $\sin(3x)$ from this section with $D = 5$ and $D = 50$ features, and with a linear model on x alone in place of the random features. What error does each reach, and why?

Adapted from (Rahimi and Recht 2007).

5. code **A clamp with a custom gradient.** Implement an `autograd.Function` whose forward is `X.clamp(lo, hi)` and whose backward passes the gradient only where the input lay strictly inside the clamp range. Compare its gradients with those of the native `torch.clamp` for inputs inside, outside, and exactly on the boundary. Which convention does the native operation use at the boundary?

   Adapted from the PyTorch tutorial Learning PyTorch with Examples and the JAX notebook Custom derivative rules.

6. [extended] **Bucketing with a straight-through estimator.** ⊠ The experiment of this section rounded the weights themselves. Here the rounding sits between two layers, on an activation. Write a layer whose forward pass maps a score s to the bucket index $\lfloor K\sigma(s) \rfloor$ and whose backward pass treats the floor as the identity. Return the index as a floating-point tensor so that gradients can reach s. Place the layer with $K = 4$ between two linear layers of output width one, and train the network on inputs $\mathbf{x} \in \mathbb{R}^{10}$ with targets $y = \lfloor 4\sigma(\mathbf{w}^\top \mathbf{x}) \rfloor$ for a fixed random $\mathbf{w}$. Compare the final training loss with that of the same network in which the floor keeps its true gradient, and explain which parameters learn in each case.

5.5 Numerical Formats

Every number in our models so far has been stored in the 32-bit floating-point format, and we have not asked whether a model needs that many bits. Fewer bits per number would save memory and time. The model with 8 billion parameters that runs through this chapter needs 32 GB to store its weights in 32 bits and 16 GB in a 16-bit format, and recent GPUs multiply 16-bit matrices at least twice as fast as 32-bit ones. Can we train in 16 bits and obtain the same model?

Not without some care. Sixteen bits make the numbers coarser, and a training run can then fail in two ways, neither of which produces an error message. In the 16-bit format fp16, 300^2 is already infinite, so a run can fill with `inf` and NaN. In the 16-bit format bf16, adding 0.001 to a weight of 1 leaves the weight at 1, so a run can stop learning.

This section shows how to compute in 16 bits without these failures. Every tensor has a *dtype*, the number format of its entries, and each format trades *range*, how large and how small its numbers can be, against *precision*, how finely it resolves them. Mixed-precision training keeps a 32-bit copy of the weights, so that small updates are not lost, and runs the expensive matrix multiplications in 16 bits. With fp16 it also scales the loss, so that small gradients do not round to zero. Section 5.2 counted the bytes per parameter of a training run; this section changes the bytes per number. Its second part, on the dtype of mixed expressions and on casting a model, is reference material and can be read quickly. The code runs on any device. The speed of 16-bit arithmetic needs a GPU with 16-bit matrix units, and Section 5.7 moves computation there.

```
import random
```

5.5.1 Floating-Point Formats

5.5.1.1 Range and Precision

Squaring 300 in the two 16-bit formats shows the two ways in which a format can fail. A value of 300 is unremarkable in a network: an unnormalized logit or an intermediate result of a variance computation can be that large.

```
x = torch.tensor(300.0)
x.to(torch.float16)**2, x.to(torch.bfloat16)**2
```

```
(tensor(inf, dtype=torch.float16), tensor(90112., dtype=torch.bfloat16))
```

The format fp16 (*half precision*) overflows to `inf`: its largest number is 65504. The format bf16 (*brain floating point*) returns 90112 instead of 90000, a finite result that is already wrong in the third digit. Both formats spend 16 bits. They merely divide them differently. A normal floating-point number is stored as a sign bit s, an exponent e, and a mantissa of t bits that encodes a fraction $f \in [0, 1)$, and its value is

$$x = (-1)^s \cdot 2^e \cdot (1 + f). \tag{5.9}$$

The exponent bits set the range of e, and so how large and how small $|x|$ can be. The mantissa bits set the precision: between 1 and 2, adjacent numbers are 2^{-t} apart, and between 2^e and 2^{e+1} they are 2^{e-t} apart, so the relative spacing is the same at every scale. The three formats of this section are:

format	sign	exponent	mantissa
fp32	1	8	23
bf16	1	8	7
fp16	1	5	10

bf16 keeps the 8 exponent bits of fp32 and cuts the mantissa to 7 bits, so it has the range of fp32 and coarse steps. fp16 keeps 10 mantissa bits and only 5 exponent bits: finer steps, much smaller range. Both outputs above can now be checked by hand. With 5 exponent bits, e runs from -14 to 15, so the largest fp16 number is $2^{15}(2 - 2^{-10}) = 65504$. With 7 mantissa bits, the bf16 numbers between 2^{16} and 2^{17} are $2^9 = 512$ apart, and 90000 lies between $175 \cdot 512 = 89600$ and $176 \cdot 512 = 90112$; it is rounded to the nearer of the two.

Figure 5.6 draws the table as bit layouts. Section 35.8 treats the floating-point representation itself in more detail.

The function `finfo` reports what each bit budget gives. Three of its numbers matter here: `max`, the largest finite value; `tiny`, the smallest *normal* value; and `eps`, the spacing 2^{-t} of representable numbers just above 1. What lies below `tiny`? There the exponent cannot shrink any further, and the format continues with *subnormal* numbers, which are evenly spaced all the way down to zero. In fp16 they are the multiples of 2^{-24}, so a value near 3×10^{-5} still keeps 9 significant bits, while one near 10^{-7} keeps a single bit.

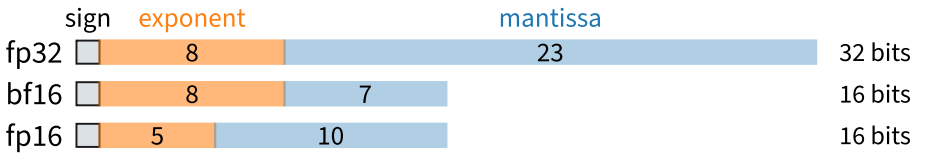


Figure 5.6: Bit layouts of the formats in the table, aligned at the sign bit, so that a longer bar means more bits: bf16 keeps the 8-bit exponent of fp32, and fp16 and bf16 split the same 16 bits in opposite ways.

```
for dtype in (torch.float32, torch.bfloat16, torch.float16):
    fi = torch.finfo(dtype)
    print(f'{str(dtype):15} max {fi.max:10.3e}  tiny {fi.tiny:9.3e}'
          f'  eps {fi.eps:8.3e}')
```

```
torch.float32   max  3.403e+38  tiny 1.175e-38  eps 1.192e-07
torch.bfloat16  max  3.390e+38  tiny 1.175e-38  eps 7.812e-03
torch.float16   max  6.550e+04  tiny 6.104e-05  eps 9.766e-04
```

bf16 has the `tiny` of fp32 and nearly the same `max`, because it has the same exponent bits. Its `eps` of 0.0078 means two to three significant decimal digits. fp16 resolves three to four digits, overflows above 65504, turns subnormal below about 6.1×10^{-5}, and has no positive number below $2^{-24} \approx 6.0 \times 10^{-8}$. Which of the two shortages is worse? Activations and gradients of a network span many orders of magnitude, and too little range turns them into `inf` or zero, while too little precision only adds rounding error. This is why bf16 is the preferred 16-bit format for training on hardware that supports it. fp16 remains in use on older GPUs and for inference.

5.5.1.2 Rounding of Small Updates

The coarse steps of bf16 matter most for the weights. An update of gradient descent changes a weight by ηg, which is often 10^{-3} or less of the weight's own size. Under round-to-nearest, an increment smaller than half the spacing of the numbers near the weight is lost entirely. We add 10^{-3} and 10^{-4} to a weight of 1 in each format:

```
w = torch.tensor(1.0)
for dtype in (torch.float32, torch.bfloat16, torch.float16):
    print(f'{str(dtype):14}: {float(w.to(dtype) + 1e-3):.7f}, '
          f'{float(w.to(dtype) + 1e-4):.7f}')
```

```
torch.float32 : 1.0010000, 1.0001000
torch.bfloat16: 1.0000000, 1.0000000
torch.float16 : 1.0009766, 1.0000000
```

In fp32 both updates register. In bf16, whose numbers near 1 are $2^{-7} \approx 0.0078$ apart, both round back to 1: the weight has not moved at all. In fp16, with spacing $2^{-10} \approx 0.00098$, the larger update survives, rounded to one step, and the smaller one is lost. A model whose weights are stored in 16 bits can therefore stop learning even though its gradients are correct. This is why training keeps the weights in fp32.

5.5.2 Dtype Promotion and Model Casting

5.5.2.1 Promotion

What dtype does an expression get when its operands disagree? We add an fp16 tensor to an fp32 tensor, to a Python float, and to an integer tensor:

```
x16 = torch.ones(3, dtype=torch.float16)
x32 = torch.ones(3, dtype=torch.float32)
(x16 + x32).dtype, (x16 + 1.0).dtype, (x16 + torch.arange(3)).dtype
```

```
(torch.float32, torch.float16, torch.float16)
```

Two float tensors promote to the wider format. This is the one rule worth remembering: an fp16 computation that meets a single stray fp32 tensor continues in fp32, without a message, and needs twice the memory from then on. Python scalars and integer tensors do not promote a float tensor, so a literal such as `x + 1.0` keeps low-precision code in low precision.

fp16 and bf16 together promote to fp32, since neither contains the other.

5.5.2.2 Casting a Model

Parameters are tensors too, so a model can be stored in 16 bits. We use the MLP of Section 4.2 throughout this section: a hidden layer of 256 ReLU units between the 784 pixels of a Fashion-MNIST image and 10 outputs, with weights drawn from $\mathcal{N}(0, 0.01^2)$ and zero biases. Its 203,530 parameters take 814,120 bytes in fp32.

```
def mlp():
    net = nn.Sequential(nn.Flatten(), nn.Linear(784, 256), nn.ReLU(),
                        nn.Linear(256, 10))
    for layer in net[1], net[3]:
        nn.init.normal_(layer.weight, std=0.01)
        nn.init.zeros_(layer.bias)
    return net

def param_bytes(net):
    return sum(p.numel() * p.element_size() for p in net.parameters())

net = mlp()
param_bytes(net)
```

```
814120
```

Layers do not promote: they require the input to have the dtype of their weights and raise an error otherwise. To change the dtype of a model we cast the whole module; `net.to(dtype)`, or the shorthand `net.bfloat16()`, converts every parameter and buffer.

```
net = net.bfloat16()
X = torch.randn(8, 1, 28, 28)
try:
    net(X)
except RuntimeError as e:
```

```
    print(e)
print(net(X.bfloat16()).dtype, param_bytes(net))
```

```
mat1 and mat2 must have the same dtype, but got Float and BFloat16
torch.bfloat16 407060
```

In every framework the parameters now take half the bytes. What happens to an fp32 input differs.

The error message shows the strictness: the fp32 input was refused rather than converted.

For *inference*, a cast model needs half the memory for its parameters; whether its predictions are still good enough must be checked for the model and task. For *training* a cast model will not do. Its weights are exactly the 16-bit weights that lost their updates in the rounding experiment; we train it below and measure what this costs.

5.5.3 Mixed-Precision Training

5.5.3.1 Master Weights and 16-Bit Compute

We seem to be stuck: the weights need 32 bits, yet the speed lies in 16. *Mixed-precision training* (Micikevicius et al. 2018) gets out of this by separating storage from computation. The parameters stay in fp32, the *master weights*, so that every update registers. The matrix multiplications of the forward and backward pass, which account for most of the computation, run in a 16-bit format, and so do the activations they produce. Reductions such as the loss are computed in fp32. Figure 5.7 shows one step.

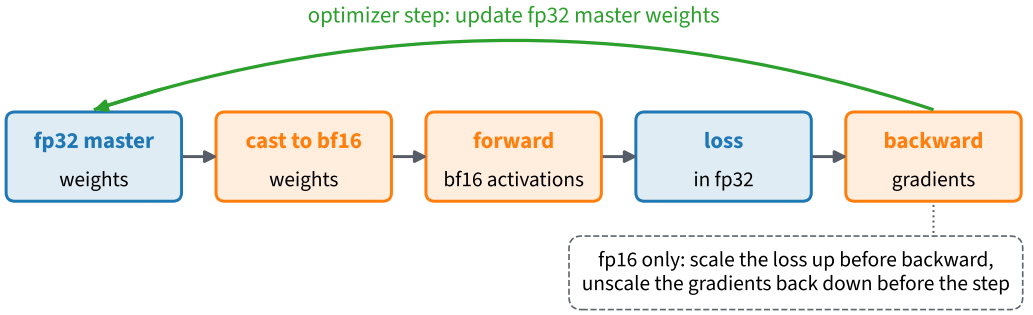


Figure 5.7: One step of mixed-precision training: the fp32 master weights are cast to bf16 for the forward pass, which produces bf16 activations; the loss is computed in fp32; the backward pass produces gradients, and the optimizer applies them to the fp32 master weights. With fp16, the loss is also scaled up before the backward pass and the gradients are scaled back down before the update.

Inside a `torch.autocast` context, every operation consults a built-in policy. Matrix multiplications and convolutions run in the 16-bit dtype, a list of sensitive operations such as `cross_entropy` runs in fp32, and other operations run in the dtype of their inputs. The parameters are not touched; each eligible operation casts its inputs as it runs.

```
net = mlp()
with torch.autocast('cpu', dtype=torch.bfloat16):
    Y = net(X)
Y.dtype, net[1].weight.dtype
```

```
(torch.bfloat16, torch.float32)
```

The output is bf16 and the weights are fp32: the opposite of `net.bfloat16()`. The first argument of `autocast` names the device type; on a GPU it is `'cuda'`.

5.5.3.2 Training in fp32 and in 16 Bits

Does 16-bit computation change what the model learns, and what does storing the weights in 16 bits cost? We train the MLP three times with plain SGD on minibatches of 256 Fashion-MNIST images: in fp32; as the cast model of the previous subsection, with its weights stored in bf16 and no fp32 copy; and with mixed precision. All three runs start from the same weights, drawn with one random seed, and see the same minibatches in the same order.

The learning rate deserves a remark. We use 0.01, a tenth of the rate at which Section 4.2 trained this MLP, and therefore 1000 steps, about four passes over the training set. The rounding experiment says why the rate matters: an update ηg is lost when it is smaller than half the spacing of the numbers near its weight. At a rate of 0.01 this holds for about 90% of the updates of the first layer, as it does late in a long run whose learning rate has been decayed. At 0.1 it holds only for the smallest 40% of them, which carry about 2% of the update, and the bf16 model would train as if nothing were wrong.

```
data = d2l.FashionMNIST(batch_size=256)
batches = list(torch.utils.data.DataLoader(data.train, batch_size=256,
                                           drop_last=True))  # One epoch
val_batch = next(iter(torch.utils.data.DataLoader(data.val, batch_size=1024)))
```

```
def train(net, mixed=False, num_steps=1000):
    dtype = next(net.parameters()).dtype  # The inputs follow the weights
    optimizer = torch.optim.SGD(net.parameters(), lr=0.01)
    losses = []
    for step in range(num_steps):
        X, y = batches[step % len(batches)]
        with torch.autocast('cpu', dtype=torch.bfloat16, enabled=mixed):
            loss = F.cross_entropy(net(X.to(dtype)).float(), y)
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()
        losses.append(loss.detach())
    Xv, yv = val_batch
    with torch.no_grad():
        accuracy = (net(Xv.to(dtype)).argmax(1) == yv).float().mean().item()
    return torch.stack(losses).tolist(), accuracy
```

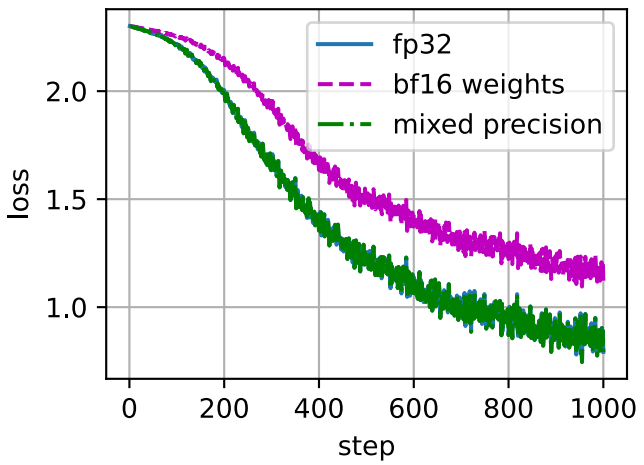
One detail of `train` has nothing to do with precision. The loss values stay on the device during training and are converted to Python numbers once, at the end, because converting every step would force the program to wait for each step to finish (Section 5.7).

`train` casts each minibatch to the dtype of the weights, so the cast model computes in bf16 throughout, while `mixed=True` keeps fp32 weights and runs the forward pass under autocast.

```
def seeded_mlp(seed):
    torch.manual_seed(seed)  # The same seed gives the same initial weights
    return mlp()

seed = random.randrange(2**31)  # One random draw, shared by all runs
loss32, acc32 = train(seeded_mlp(seed))
loss_bf16, acc_bf16 = train(seeded_mlp(seed).bfloat16())
loss_mixed, acc_mixed = train(seeded_mlp(seed), mixed=True)
print(f'validation accuracy: fp32 {acc32:.3f}, bf16 weights {acc_bf16:.3f}, '
      f'mixed {acc_mixed:.3f}')
d2l.plot(list(range(1, 1001)), [loss32, loss_bf16, loss_mixed], 'step',
         'loss', legend=['fp32', 'bf16 weights', 'mixed precision'])
```

```
validation accuracy: fp32 0.681, bf16 weights 0.652, mixed 0.681
```



The loss curves of the fp32 run and of mixed precision lie on top of each other, and their validation accuracies agree to within about one percentage point. The run with bf16 weights falls behind from the first hundred steps on: after 1000 steps its loss is about 1.15 against 0.85, and it classifies about 64% of the validation images correctly against about 68%. Most of its first-layer updates are rounded away, as in the rounding experiment, while mixed precision applies the same updates to fp32 master weights and loses none of them. None of the three models is fully trained; the comparison needs only that the runs see the same data for the same number of steps.

What have we gained? Not speed, in this experiment: the speed of 16-bit arithmetic depends on matrix units that compute in 16 bits, as the tensor cores of recent GPUs do, and on models large enough to keep them busy, and Section 5.7.2.1 measures it on a GPU.

Note, however, what mixed precision does not shrink. The master weights and the optimizer state remain fp32, so the bytes per parameter counted in Section 5.2 stay exactly as they were. The 16 GB of the opening are what the 8-billion-parameter model occupies once it is cast for inference. To train it, mixed precision saves only the memory of the activations, which are stored in 16 bits, and the time of the matrix multiplications.

5.5.3.3 Loss Scaling for fp16

With bf16, the recipe above is complete. fp16 fails at the other end of its range as well: small gradients *underflow*. Below $2^{-14} \approx 6.1 \times 10^{-5}$ they are stored as subnormals and lose significant bits, and below about 3×10^{-8}, half the smallest subnormal, they are rounded to zero. A gradient of 10^{-8} vanishes, but the same gradient multiplied by 2^{16} is stored without trouble:

```
g = torch.tensor(1e-8)
g.half(), (g * 2**16).half()
```

```
(tensor(0., dtype=torch.float16), tensor(0.0007, dtype=torch.float16))
```

How many real gradients are that small? We compute the gradient of the loss with respect to the first-layer weights of the MLP, on the first minibatch, in fp32, and count the nonzero entries that fp16 could store only as subnormals or zeros, with and without a factor of 2^{16}:

```
net = mlp()
X, y = batches[0]
F.cross_entropy(net(X), y).backward()
g = net[1].weight.grad.flatten()
g = g[g != 0]
for scale in (1, 2**16):
    print(f'scale {scale:5}: '
          f'{((g * scale).abs() < 2**-14).float().mean():.1%} subnormal, '
          f'{((g * scale).half() == 0).float().mean():.2%} zero')
```

```
scale     1: 32.1% subnormal, 0.21% zero
scale 65536: 0.0% subnormal, 0.00% zero
```

About a third of the nonzero first-layer gradients lie in the subnormal range of fp16, and about two in a thousand would round to zero. For this small network the damage is slight, since the entries that vanish are the smallest ones. We should be candid here: we found no small example in which a run fails in fp16 and the remedy below rescues it. Real damage needs nearly all of a gradient below the threshold, which can happen when a loss is averaged over many elements or a gradient has passed through many layers. Micikevicius et al. (2018) report that many networks, their ImageNet classifiers among them, trained in fp16 without any remedy, while some did not: an object detector diverged, and it and a large recurrent language model matched their fp32 accuracy only with the remedy that follows.

Multiplied by 2^{16}, none of our gradients is subnormal, and none comes near the overflow threshold of 65504 either. This is the idea of *loss scaling*. We multiply the loss by a large constant S before the backward pass, and because differentiation is linear, every gradient is multiplied by S as well. Before the update, we divide the gradients by S in fp32, where the small values are representable. The update is unchanged; only the gradients in between are kept in a range that fp16 can store.

A fixed S that is too large makes large gradients overflow to `inf`. The standard remedy chooses S dynamically: start high, and whenever the scaled gradients contain `inf` or NaN, skip that update and halve S; after a run of clean steps, double it again.

`torch.amp.GradScaler` implements dynamic loss scaling. The loop below trains the MLP for 20 steps with fp16 autocast:

```
net = mlp()
optimizer = torch.optim.SGD(net.parameters(), lr=0.1)
scaler = torch.amp.GradScaler('cpu')
for X, y in batches[:20]:
    with torch.autocast('cpu', dtype=torch.float16):
        loss = F.cross_entropy(net(X), y)
    optimizer.zero_grad()
    scaler.scale(loss).backward()  # Gradients of S * loss
    scaler.step(optimizer)         # Divide by S, skip the step on inf/NaN
    scaler.update()                # Adjust S
print(f'loss {loss.item():.3f}, loss scale {scaler.get_scale():.0f}')
```

```
loss 2.032, loss scale 65536
```

The scale starts at 2^{16} and stays there, because no step overflowed.

On hardware that supports bf16, which includes recent GPUs, TPUs, and recent CPUs, the standard choice is therefore bf16 mixed precision without loss scaling. fp16 with loss scaling remains for older GPUs.

A last shortcut changes fp32 computation without changing any dtype. On NVIDIA GPUs since the Ampere generation, an fp32 matrix multiplication can run in *tf32* mode on the 16-bit matrix units: its inputs are rounded to the 10-bit mantissa of fp16, keeping the 8-bit exponent of fp32, and the products are summed in fp32, so the result is faster and typically agrees with full fp32 to three or four significant digits.

PyTorch computes fp32 matrix multiplications in full fp32 by default; `torch.set_float32_matmul_precision('high')`, which the imports of Section 5.7 set, allows tf32.

5.5.4 Numerical Failure Modes

Three failures account for most numerical trouble in 16-bit code: an overflow that later appears as NaN, an exponential that overflows before the result does, and a long sum that loses its small terms.

5.5.4.1 From Overflow to NaN

A NaN in the loss is often the second symptom. The first is an overflow to `inf`, and arithmetic on infinite values produces NaN:

```
s = torch.tensor(60000., dtype=torch.float16) * 2  # Overflows
s, s - s
```

```
(tensor(inf, dtype=torch.float16), tensor(nan, dtype=torch.float16))
```

The overflow can happen many operations before the NaN reaches the loss, and once an update contains NaN, every later result does too. When a loss turns NaN, check ranges first: does any computation run in fp16, and do intermediate values approach its maximum of 65504? Only then look for other causes, such as a learning rate that is too large.

Under autocast with `GradScaler`, the check for `inf` and NaN in the gradients catches an overflow in the step where it occurs and skips the update.

5.5.4.2 Log-Sum-Exp

The cross-entropy of a softmax output contains the logarithm of a sum of exponentials of the logits. The direct evaluation of $\log \sum_i \exp(x_i)$ overflows long before its result does:

```
x = torch.tensor([12.0, 11.0, 10.0], dtype=torch.float16)
x.exp().sum().log(), torch.logsumexp(x, dim=0)
```

```
(tensor(inf, dtype=torch.float16), tensor(12.4062, dtype=torch.float16))
```

The result is about 12.4, but the intermediate value $e^{12} \approx 163{,}000$ exceeds 65504. The remedy is the one we used for the softmax in Section 3.4.2: subtract the largest entry m before exponentiating and add it back afterwards,

$$\log \sum_i \exp(x_i) = m + \log \sum_i \exp(x_i - m). \tag{5.10}$$

No exponent on the right is positive, so nothing overflows, and the largest term equals 1, so the sum cannot underflow to zero either. Nobody needs to write the shift by hand: the log-softmax and cross-entropy functions of every framework apply it.

So does `torch.logsumexp`, which the cell calls.

5.5.4.3 Accumulation in fp32

A long sum in a 16-bit format drifts: once the running total is large, each small increment falls below half the spacing of the numbers near the total and is rounded away. We add 2^{-10}, which fp16 stores exactly, 4096 times from left to right, once into an fp16 running total and once into an fp32 running total:

```
total16, total32 = np.float16(0), np.float32(0)
term = np.float16(2**-10)
for _ in range(4096):
    total16 += term  # fp16 accumulator
    total32 += term  # fp32 accumulator, same fp16 terms
total16, total32
```

```
(np.float16(2.0), np.float32(4.0))
```

The exact sum is 4, and the fp32 total reaches it. The fp16 total stops at 2, after 2048 additions: between 2 and 4 the fp16 numbers are 2^{-9} apart, so $2 + 2^{-10}$ lies exactly halfway between two of them, and round-to-nearest-even returns 2, whose last mantissa bit is even. Means over large batches, loss totals over an epoch, and variances all follow this pattern, which is why mixed-precision recipes compute reductions and normalization layers in fp32. When you write a reduction over 16-bit data yourself, give it an fp32 accumulator.

These failures share one cause: a 16-bit format has too little range or too little precision for one intermediate value, even when the inputs and the result are representable. Mixed precision also changes results in their last bits by design, so a run with tf32 or mixed precision never matches an fp32 run bit for bit. Which runs can be expected to agree exactly, and how to make them agree, is the subject of Section 5.8.

5.5.5 Summary

A floating-point format divides its bits between the exponent, which sets its range, and the mantissa, which sets its precision. Of the two 16-bit formats, fp16 has more precision and much less range, while bf16 has the range of fp32 and only two to three significant digits. Stored in 16 bits, weights lose updates that are small relative to their size, so mixed-precision training keeps fp32 master weights and runs the matrix multiplications and activations in 16 bits; on the MLP, its loss curve matches that of fp32 training, while the same model with its weights stored in bf16 falls behind. fp16 additionally needs loss scaling, which multiplies the loss and divides the gradients by the same factor so that small gradients do not underflow; bf16 does not. Compute exponentials in their shifted form and long sums in an fp32 accumulator.

In PyTorch, `net.bfloat16()` casts a model, `torch.autocast` runs eligible operations in 16 bits over fp32 parameters, and `torch.amp.GradScaler` scales the loss for fp16.

5.5.6 Exercises

1. **Rounded updates.** A weight $w = 1$ receives 1000 updates of $+10^{-3}$ each and is rounded to its storage dtype after every update.
    1. Using the `eps` values of this section and round-to-nearest, predict the final value of w to three decimal places when it is stored in fp32, in fp16, and in bf16.
    2. Now the updates go to an fp32 master copy, and a bf16 working copy is rounded from the master copy after every update. Predict the final values of both copies.
2. **Mixed-precision memory.** Training with Adam in fp32 stores 16 bytes per parameter (Section 5.2). Suppose that `train` used Adam instead of SGD.
    1. In which dtype does the mixed-precision recipe of your framework store the weights, the gradients, and Adam's running averages between steps? Does mixed precision reduce the bytes per parameter?
    2. Count as stored activations the input of each linear layer and the logits. For the MLP of `train`, convert the per-example count to bytes in fp32 and in the 16-bit format `train` uses. At which batch size does the activation saving equal the persistent training state of this MLP?
3. **Loss scaling on overflow.** Under dynamic loss scaling (Section 5.5.3.3), a step whose unscaled gradients contain an `inf` or NaN is skipped and the scale is halved. After N consecutive clean steps, the scale is doubled.
    1. A skipped step must leave Adam's running averages untouched. Explain what would go wrong if the step instead updated them with the non-finite gradient, and what would go wrong if it updated them with the non-finite entries replaced by zeros.
    2. Explain why the scale must grow again after a run of clean steps instead of only shrinking.
    3. Suppose that every step overflows when the scale is above a fixed threshold and that no step overflows below it. Once the scale has settled, what fraction of steps is skipped as a function of N? Without the assumption of a fixed threshold, what does a larger N cost?
4. code **Normalization in fp16.** Mixed-precision recipes run normalization layers in fp32. To see why, consider RMSNorm with its gain set to one (Section 5.4 implements

it as a module). It rescales an input row $\mathbf{x}$ of width d by its root mean square,

$$\frac{\mathbf{x}}{\sqrt{\frac{1}{d}\sum_{i=1}^{d} x_i^2 + \epsilon}}, \tag{5.11}$$

where the small constant ϵ guards against division by zero. Compute it entirely in fp16 for a batch of 64 rows of width $d = 1024$ whose entries have a standard deviation of about 100.

1. Which intermediate quantity is the first to become `inf`, and what does the layer return for the affected rows?
2. Take the squares as exact. At what input standard deviation does their mean exceed the fp16 maximum, and at what standard deviation does their sum over the d entries exceed it?

5. code **The learning rate at which bf16 weights fail.** Give `train` a learning-rate argument and repeat the fp32 run and the run with bf16 weights at the rates 0.1, 0.03, 0.01, and 0.003, with the number of steps chosen so that the rate times the number of steps stays at 10.

   1. At which rate does the run with bf16 weights begin to fall behind?
   2. A first-layer weight of this MLP has a size of about 0.01. What is the bf16 spacing near it, and how small must a gradient entry be at each rate for its update to be lost?

6. code **The e4m3fn format.** Print `torch.finfo(torch.float8_e4m3fn)` and compare its `max`, `tiny` and `eps` with those of fp16 and bf16. Explain the name `e4m3fn`. A format with a 4-bit exponent and a 3-bit mantissa that followed the IEEE convention would reserve the all-ones exponent for `inf` and NaN. Compute that format's largest finite value, and explain how `e4m3fn` reaches a `max` of 448 instead.

5.6 Saving, Loading, and Pretrained Weights

A trained model is a fragile thing: it lives only in the memory of the process that trained it. When the process ends, because the script finished, crashed, or ran on a cloud machine that was reclaimed, the weights are gone unless they were written to disk. For the small models of earlier chapters this is a nuisance: a few minutes of training to repeat. For a large model it can mean weeks of computation, and at scale interruptions are the rule. In a 54-day snapshot of the pre-training of Llama 3 405B on up to 16,384 GPUs, Meta recorded 419 unexpected interruptions, about one every three hours (Grattafiori et al. 2024). A run that cannot continue from where it stopped would never finish.

Saved weights also let others use a model without training it: in practice, many models start from weights that someone else trained, as the fine-tuning examples of Section 32.2 do. Both uses raise the same questions: what must be written to disk, and how is it read back?

What has to be saved follows from Section 5.2. A model is code plus state. The code defines the layers and the forward computation; the *state* is the set of named tensors that training produced, parameters and buffers alike. We save only the state, as a mapping from names to tensors, and restore a model by building its structure from code and loading the tensors into

it. Resuming training needs more than the weights: it also needs the optimizer's running averages, the random-number streams, and the step counter, which makes the file several times larger.

This is a practical section, about files: what goes into them, which of them can be trusted, and what a load must check. We first save and load the state of a small network. We then ask which file formats are safe to load from a stranger, write a checkpoint from which training resumes exactly, and load pretrained weights into a network with a new output layer.

```
import json
import os
import struct
from collections import Counter
from dataclasses import asdict, dataclass
import numpy as np
import torch
from torch import nn
from torch.nn import functional as F
from torchvision.models import resnet50, ResNet50_Weights
from safetensors.torch import load_file, save_file
```

5.6.1 Model State

A model holds many tensors, so a file format for its state must store a collection of named tensors. The framework builds that collection for us. Its names are the paths through the module tree of Section 5.1, and its values are the tensors. Here is the state of an MLP with one hidden layer of 256 units.

`state_dict` maps the dotted paths (`hidden.weight`, `output.bias`) to the tensors, buffers included.

```
class MLP(nn.Module):
    def __init__(self):
        super().__init__()
        self.hidden = nn.Linear(20, 256)
        self.output = nn.Linear(256, 10)

    def forward(self, x):
        return self.output(F.relu(self.hidden(x)))

net = MLP()
X = torch.randn(2, 20)
Y = net(X)
{name: tuple(t.shape) for name, t in net.state_dict().items()}
```

```
{'hidden.weight': (256, 20),
 'hidden.bias': (256,),
 'output.weight': (10, 256),
 'output.bias': (10,)}
```

The state holds names, shapes, and values, and that is all. It does not record the `MLP` class, its `forward` computation, or the activation function. Loading it therefore needs code that

builds a model with the same names and shapes. Renaming an attribute, say `output` to `head`, changes the names and breaks the match even though the computation stays the same. The last part of this section shows how to detect such mismatches when loading.

5.6.2 Pickle and safetensors

A file that stays on our own disk can be in any format. A file downloaded from a model hub, or received from a colleague, raises a question that tensors on our own disk do not: can loading it do anything besides creating tensors? Unfortunately, the answer depends on the format.

5.6.2.1 Code Execution on Load

Python's standard serialization, `pickle`, does not store data. It stores a program that reconstructs the data, and loading the file runs that program. The program may call any function the loading process can import. An object tells pickle what to call through its `__reduce__` method, which returns a function and its arguments. The class below asks for `print`; a malicious file would ask for something like `os.system`.

```
class Tripwire:
    def __reduce__(self):  # pickle calls print(...) to rebuild the object
        return (print, ('*** payload executed while loading ***',))
```

`torch.save` writes pickle. Loading the file with `weights_only=False`, the behavior of `torch.load` before PyTorch 2.6, runs the payload:

```
torch.save(Tripwire(), 'tripwire.pt')
_ = torch.load('tripwire.pt', weights_only=False)  # the payload runs here
```

```
*** payload executed while loading ***
```

Since version 2.6, `torch.load` defaults to `weights_only=True`, which accepts only tensors, numbers, strings, and containers of them, and refuses the file:

```
try:
    torch.load('tripwire.pt')  # weights_only=True, the default
except Exception as e:
    print(type(e).__name__, str(e).split('.')[0])
```

```
UnpicklingError Weights only load failed
```

A refusing loader protects us only as long as its checks are complete, and such checks have had holes, in PyTorch's `weights_only` loader among others. A format that contains no program at all removes the question.

5.6.2.2 The safetensors Format

The *safetensors* format contains no program: a file holds tensors and a description of them, and nothing else. It has three parts, shown in Figure 5.8: an 8-byte little-endian integer that gives the length of a header, the header itself as JSON text, and the raw bytes of all tensors back to back. The header names each tensor and records its dtype, its shape, and the range

of bytes it occupies after the header. Nothing in the file is executed when it is loaded; a loader only has to check that the header is well formed and that the ranges lie inside the file. Because the ranges are known in advance, a reader can map the file into memory and read one tensor without reading the others. The format does not depend on any framework, and it is the default format for weights on the Hugging Face Hub.

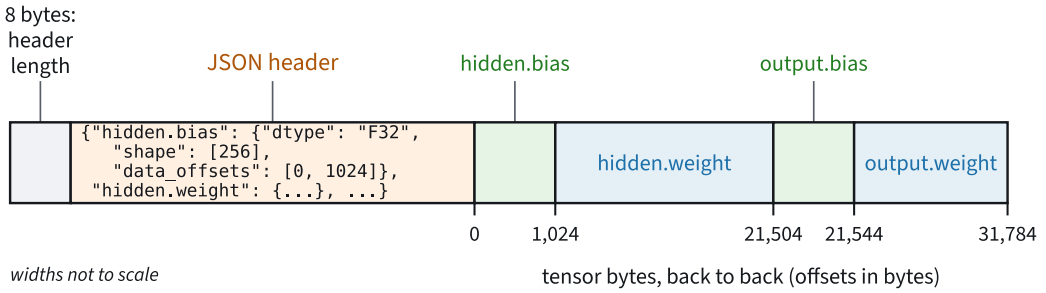


Figure 5.8: The safetensors file of the MLP below as one strip of bytes: an 8-byte length, the JSON header with each tensor's dtype, shape, and byte range, and the raw bytes of the four tensors back to back, in the order of their names. The offsets under the data are the ranges that the header records.

We save the state of the MLP as a safetensors file, load it into a second instance, and check that the two networks compute the same outputs.

`save_file` takes the `state_dict` directly, and `load_file` returns one.

```
save_file(net.state_dict(), 'mlp.safetensors')
clone = MLP()
clone.load_state_dict(load_file('mlp.safetensors'))
torch.equal(clone(X), Y)
```

```
True
```

The round trip is exact. Because the header is JSON at a known offset, we can read it with nothing but the Python standard library:

```
with open('mlp.safetensors', 'rb') as f:
    n = struct.unpack('<Q', f.read(8))[0]  # header length, little-endian
    header = json.loads(f.read(n))
print('header length:', n, 'bytes')
for name, entry in sorted(header.items(), key=lambda e: e[1]['data_offsets']):
    print(f"{name:18s} {entry['dtype']} {str(entry['shape']):10s} "
          f"bytes {entry['data_offsets']}")
```

```
header length: 296 bytes
hidden.bias        F32 [256]      bytes [0, 1024]
hidden.weight      F32 [256, 20]  bytes [1024, 21504]
output.bias        F32 [10]       bytes [21504, 21544]
output.weight      F32 [10, 256]  bytes [21544, 31784]
```

Each byte range is the number of entries times 4 bytes, since the dtype `F32` is float32. The

bias of the hidden layer has 256 entries and occupies the first 1,024 bytes of the data; the weight matrix of the hidden layer, with $256 \times 20 = 5{,}120$ entries, takes the next 20,480 bytes. Every range starts where the previous one ends, so the data contains nothing but the tensors.

For files that stay on our own machine, the framework's native format serves well. For files that cross a trust boundary, in either direction, we use a format that contains no program and publish a checksum with the file so that the recipient can detect a corrupted or replaced download.

5.6.3 Training Checkpoints

A file of weights is enough to use a model, but not to continue training it. The reason is that the optimizer may have a memory of its own. Many of the later chapters train with *Adam* (Section 9.5), which keeps two running averages for every parameter (Section 5.2): $\mathbf{m}$, of its gradient $\mathbf{g}$, and $\mathbf{v}$, of the gradient's elementwise square. Each parameter moves against its averaged gradient, by an amount scaled down where its gradients have been large:

$$\mathbf{w} \leftarrow \mathbf{w} - \eta \frac{\hat{\mathbf{m}}}{\sqrt{\hat{\mathbf{v}}} + \epsilon}. \tag{5.12}$$

Here η is the learning rate, ϵ a small constant such as 10^{-8}, and the division and square root act elementwise. The hats mark a rescaling that corrects for the start of both averages at zero; its formula can wait for Section 9.5. What matters here is that $\mathbf{m}$ and $\mathbf{v}$ carry the history of the run: drop them, and the optimizer behaves as if training had just begun.

A checkpoint from which training can resume therefore holds five things (Figure 5.9): the model state; the optimizer state, here $\mathbf{m}$ and $\mathbf{v}$; the state of the random-number generators, so that data shuffling and dropout continue the same streams; the step counter, which Adam's correction uses and which also sets the position in any learning-rate schedule; and the configuration that sizes the model when it is rebuilt. Gradients are not among them, since the next step recomputes them. This leaves 12 of the 16 bytes per parameter that training with Adam in fp32 stores (Section 5.2): the weights and the two averages. For a model with 8 billion parameters the checkpoint therefore takes 96 GB, three times the 32 GB of its fp32 weights and six times the 16 GB file that is enough for inference in bf16 (Section 5.5).

A checkpoint must also survive a crash while it is being written. If the new file replaces the old one in place, a crash in the middle leaves neither. We therefore write to a temporary name and rename the finished file; a rename within one file system is atomic, so a reader sees either the old checkpoint or the new one.

The helpers below keep the checkpoint to tensors, numbers, and strings, so that it loads with `weights_only=True`. This is why the configuration, a dataclass, is stored as a dictionary through `asdict`. Besides the global random-number streams, the helpers save and restore a `torch.Generator` passed as `rng`, such as the one that will draw our minibatches. The rename is `os.replace`.

```
def save_checkpoint(path, model, optimizer, step, cfg=None, rng=None):
    """Atomically write a resumable training checkpoint."""
    ckpt = {'model': model.state_dict(),
            'optimizer': optimizer.state_dict(),
```

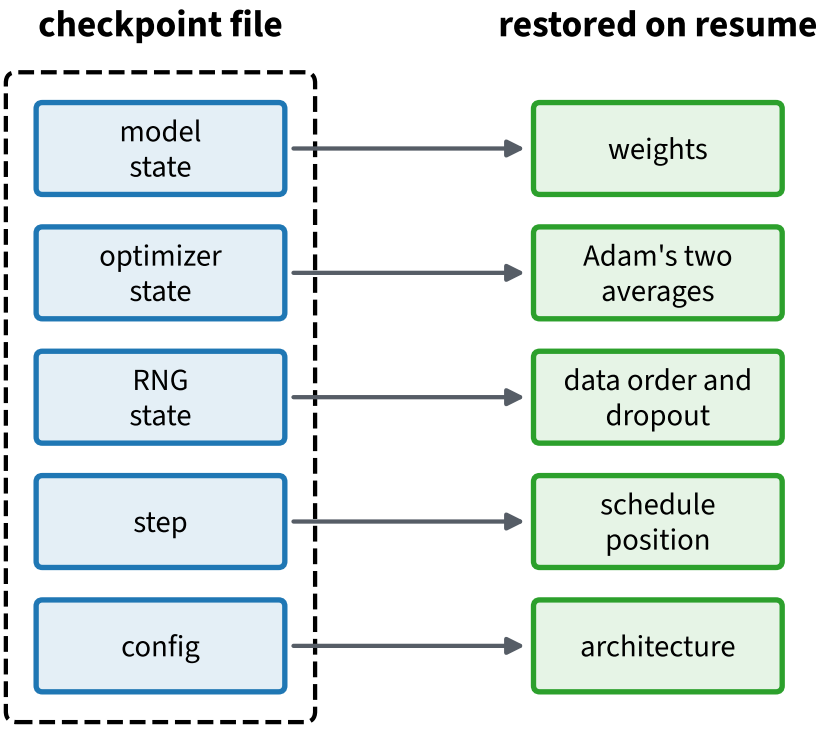


Figure 5.9: A checkpoint file's five compartments, each paired by an arrow with what it restores on resume: model state with the weights, optimizer state with Adam's two running averages, RNG state with the data order and dropout, step with the position in the schedule, and config with the architecture.

```
            'step': step,
            'cpu_rng': torch.get_rng_state()}
    if cfg is not None:
        ckpt['cfg'] = asdict(cfg)
    if rng is not None:                          # a torch.Generator
        ckpt['rng'] = rng.get_state()
    if torch.cuda.is_available():
        ckpt['cuda_rng'] = torch.cuda.get_rng_state_all()
    tmp = path + '.tmp'
    torch.save(ckpt, tmp)
    os.replace(tmp, path)

def load_checkpoint(path, model, optimizer=None, rng=None):
    """Restore the state written by save_checkpoint; return the raw dict."""
    ckpt = torch.load(path, weights_only=True)
    model.load_state_dict(ckpt['model'])
    if optimizer is not None:
        optimizer.load_state_dict(ckpt['optimizer'])
    if rng is not None:
        rng.set_state(ckpt['rng'])
    torch.set_rng_state(ckpt['cpu_rng'])
    if torch.cuda.is_available() and 'cuda_rng' in ckpt:
        torch.cuda.set_rng_state_all(ckpt['cuda_rng'])
    return ckpt
```

The experiment trains a small regressor: an MLP with one hidden layer of 64 units, fit by Adam with learning rate 0.05 to 256 examples of a noisy linear function of 20 inputs. Each step computes the gradient on a minibatch of 64 examples drawn at random. The draws come from a random-number generator of their own, seeded with 0, so the order of the minibatches is part of the state that a checkpoint must save. The configuration is a dataclass, the form used in Section 5.1; it is what a rebuild reads to size the model, so it goes into the checkpoint.

```
@dataclass
class Config:
    num_inputs: int = 20
    num_hiddens: int = 64
    num_outputs: int = 1
    batch_size: int = 64
    lr: float = 0.05
```

We train for 200 steps, save a checkpoint, and then let the same run go on for 5 more steps, as if nothing had happened. These 205 uninterrupted steps are the reference that a resumed run must match. The function `train` returns the loss of each minibatch, computed before the step that uses it.

```
def build(cfg):
    net = nn.Sequential(nn.Linear(cfg.num_inputs, cfg.num_hiddens),
                        nn.ReLU(),
                        nn.Linear(cfg.num_hiddens, cfg.num_outputs))
    for layer in (net[0], net[2]):
```

```
        nn.init.xavier_uniform_(layer.weight)
        nn.init.zeros_(layer.bias)
    return net

cfg = Config()
data = torch.randn(256, cfg.num_inputs)
target = data @ torch.randn(cfg.num_inputs, 1) + 0.1 * torch.randn(256, 1)
loss = nn.MSELoss()

def train(model, opt, rng, num_steps):
    losses = []
    for _ in range(num_steps):
        idx = torch.randint(len(data), (cfg.batch_size,), generator=rng)
        opt.zero_grad()
        l = loss(model(data[idx]), target[idx])
        l.backward()
        opt.step()
        losses.append(l.detach())
    return [l.item() for l in losses]

net = build(cfg)
opt = torch.optim.Adam(net.parameters(), lr=cfg.lr)
rng = torch.Generator().manual_seed(0)      # draws the minibatches
train(net, opt, rng, 200)
path = 'run.pt'
save_checkpoint(path, net, opt, step=200, cfg=cfg, rng=rng)
uninterrupted = train(net, opt, rng, 5)
```

To resume, a new process builds the model and the optimizer from the saved configuration, seeds its generator as at the start of the run, loads the checkpoint, and reads from the saved step how many steps remain. We resume twice and compare the final weights with those of the uninterrupted run: once from the full checkpoint, and once without the generator's state, as a script would that restores everything else but simply seeds its generator again.

`load_state_dict` overwrites the tensors of the model and the optimizer in place. It raises an error if a name or a shape in the file does not match the model.

```
def resume(path, num_steps, restore_optimizer=True, restore_rng=True):
    cfg = Config(**torch.load(path, weights_only=True)['cfg'])  # saved config
    model = build(cfg)
    opt = torch.optim.Adam(model.parameters(), lr=cfg.lr)
    rng = torch.Generator().manual_seed(0)  # seeded as at the start
    ckpt = load_checkpoint(path, model, opt if restore_optimizer else None,
                           rng=rng if restore_rng else None)
    return model, train(model, opt, rng, num_steps - ckpt['step'])

def max_diff(a, b):
    return max((p - q).abs().max().item()
               for p, q in zip(a.parameters(), b.parameters()))
```

```
for name, kwargs in (('full checkpoint', {}),
                     ('without the generator', {'restore_rng': False})):
    resumed, _ = resume(path, 205, **kwargs)
    print(f'{name:22s} max |difference| {max_diff(resumed, net):.3g}')
```

```
full checkpoint        max |difference| 0
without the generator  max |difference| 0.00397
```

Resumed from the full checkpoint, the run ends with exactly the weights of the run that was never interrupted: the largest difference is 0, bit for bit. Without the generator's state, the resumed steps draw the first minibatches of the run again instead of the next ones, and the weights differ by roughly 0.005 to 0.02. The step counter told both resumed runs where to stop. Bitwise equality also needs computations that give the same result every time they run, a topic that Section 5.8 takes up.

A missing optimizer state does more damage, and the loss shows it. We resume once more with the saved weights and generator but a newly created Adam, and print the minibatch losses of the five steps next to those of the full resume.

```
_, full = resume(path, 205)
_, fresh = resume(path, 205, restore_optimizer=False)
print('saved optimizer:', [round(l, 4) for l in full])
print('fresh optimizer:', [round(l, 4) for l in fresh])
```

```
saved optimizer: [0.0009, 0.0017, 0.0022, 0.001, 0.0019]
fresh optimizer: [0.0009, 2.1294, 0.6979, 0.7146, 0.9511]
```

With the saved optimizer state, training continues where it stopped, and the minibatch losses stay at their level at the checkpoint, about 0.01 or less. With a fresh Adam, a single step raises the loss by a factor of several hundred or more, and four steps later it has not recovered: it is still a hundred times higher or more.

How can one step of an optimizer do so much damage to a model that was nearly done? Equation 5.12 explains it. A fresh Adam starts with $\mathbf{m}$ and $\mathbf{v}$ at zero. After one update each holds a small fraction of $\mathbf{g}$ and of $\mathbf{g} \odot \mathbf{g}$, and the correction divides exactly that fraction out, which leaves $\hat{\mathbf{m}} = \mathbf{g}$ and $\hat{\mathbf{v}} = \mathbf{g} \odot \mathbf{g}$. The step of each parameter then has size $\eta\,|g_i|/(|g_i| + \epsilon)$, almost exactly η. Every parameter with a nonzero gradient moves by the full learning rate of 0.05, however small that gradient is. The saved $\mathbf{v}$, by contrast, still holds the larger squared gradients of earlier steps, so $\sqrt{\hat{\mathbf{v}}}$ is large compared with the current gradients, and the steps stay small. The closer a run is to its minimum, the smaller its gradients and the larger the damage of a step that ignores their size. Weights alone do not resume a run.

5.6.4 Pretrained Weights

We now turn to the second use of a saved state: starting from weights that someone else trained. A network trained on a large dataset such as ImageNet, 1.2 million images in 1,000 classes, computes features that are useful for other image tasks too. To reuse it for a task with 10 classes, we keep all layers except the last and replace that one, the *head*, with a new layer of 10 outputs. Our example is ResNet-50, a convolutional network with about 25 million parameters (23.5 million without its 1,000-class head), trained on ImageNet. We

have not met convolutional networks yet, and luckily we do not need to: for loading weights, ResNet-50 is simply a model state of a few hundred named tensors.

torchvision downloads the weights the first time we pass a `weights=` value.

```
net = resnet50(weights=ResNet50_Weights.DEFAULT)  # ~100 MB once
net.fc = nn.Linear(net.fc.in_features, 10)        # new 10-class head
net.fc
```

```
Linear(in_features=2048, out_features=10, bias=True)
```

Weights from elsewhere often do not fit our model exactly, and a loader that fills what it can without saying what it skipped hides the mismatch. To see what a careful load reports, we remove two groups of entries from the pretrained state. The first is the head, which we mean to replace anyway. The second is the last of the four stages into which ResNet-50 groups its layers, and it stands in for an accident: a file that lost part of its contents. We load what remains into a new, randomly initialized ResNet-50 with a 10-class head, and compute two sets of names. *Missing* names are parameters of the model that the file did not fill; they keep their random initialization. *Unexpected* names are entries of the file that have no place in the model.

`load_state_dict(..., strict=False)` loads what matches and returns the two sets as `missing_keys` and `unexpected_keys` instead of raising an error.

```
damaged = {k: v for k, v in net.state_dict().items()
           if not k.startswith(('fc.', 'layer4.'))}
target = resnet50(weights=None)
target.fc = nn.Linear(target.fc.in_features, 10)
report = target.load_state_dict(damaged, strict=False)
print('missing by block:', dict(Counter(
    k.split('.')[0] for k in report.missing_keys)))
print('unexpected:', report.unexpected_keys)
```

```
missing by block: {'layer4': 50, 'fc': 2}
unexpected: []
```

The two entries of the head, its weight and its bias, are expected: the head is new and is meant to start random. The entries of the last stage are not. Had we loaded without looking, a large part of the network would now compute its features with random weights, and no error would have told us. The set of unexpected names is empty here; a nonempty one usually means that a layer was renamed, so that its weights found no place in the model. In practice, we state in advance which names may be missing, the head here, and treat every other entry of either set as an error.

Training on the new task then usually starts by *freezing* the pretrained layers, by the means of Section 5.2.3.2, so that the optimizer changes only the new head. Whether and how far to unfreeze them later is a question for Section 32.2.

All of this assumed that the state lives in main memory, and the computation with it. Section 5.7 moves both to GPUs.

5.6.5 Summary

A saved model is state, not code: a mapping from names to tensors that becomes a model only when code rebuilds a structure with the same names and shapes. Loading a pickle file runs a program, so files from elsewhere belong in a format without one, such as safetensors, which stores a JSON header and raw bytes. A checkpoint that resumes training adds the optimizer state, the random-number state, the step, and the configuration, and is written to a temporary name and renamed into place. Resumed from all of them, our run ended with the weights of the uninterrupted run, bit for bit. Without the state of the generator that draws the minibatches, the two runs drift apart. Without the optimizer state, Adam's first step moves every parameter by the full learning rate, and the loss of a model near its minimum jumps by orders of magnitude. Pretrained weights are loaded by name into a model with a new head; the names that the file did not fill, and the file entries that found no place, must be read and checked against what was expected.

5.6.6 Exercises

1. **Reasons to checkpoint.** Name two reasons to checkpoint a run that never leaves the machine it trains on, and state for each which compartments of Figure 5.9 it needs.

2. code **Safetensors header.** The header length of the MLP's safetensors file is the 8-byte little-endian integer at the start of the file (Figure 5.8).

    1. Predict, then measure, how that length changes when the hidden width is doubled and when a third layer is added. Explain which parts of the header depend on tensor sizes at all.
    2. Read the hidden layer's bias from the file using only the header and file seeks, without the safetensors library. Decode the bytes with the dtype and shape that the header records, and check that the result equals the tensor that `load_file` returns.
    3. Explain why a pickle-based checkpoint cannot offer the same single-tensor read without deserializing the whole object graph.

3. code **Partial loading.** Define `MLP2`, a copy of `MLP` that inserts a second hidden layer `hidden2` of width 256 before the output layer and renames the attribute `output` to `head`.

    1. Load `mlp.safetensors` into `MLP2` with `load_state_dict(..., strict=False)`. Read the returned report and explain each missing and each unexpected key.
    2. Rename the file's keys to the names that `MLP2` uses before loading. Confirm that only `hidden2` keeps its fresh initialization, and make the load raise on any unexpected key and on any missing key outside `hidden2`.

    Adapted from Matthew Inkawhich, Saving and Loading Models, "Warmstarting Model Using Parameters from a Different Model".

4. code **Dtype round trip.** ▯ A `float16` number stores 5 exponent bits and 10 mantissa bits, against 8 and 23 in `float32` (Section 5.5). Cast the parameters of an `MLP` to `float16`, save them, and load them back into a `float32` `MLP`.

    1. Predict the largest relative error of a restored parameter from the mantissa width. Measure the maximum absolute difference and, over the nonzero parameters, the maximum relative difference between the restored and the original values. Explain any parameter whose relative error exceeds the prediction.

2. Compare the outputs of the original and the restored `MLP` on `X`, and decide whether the round trip is acceptable for inference.
3. Adam keeps a running average of squared gradients. Compute what `float16` stores for the square of a gradient entry of 10^{-4}, and explain why a resumable checkpoint keeps the optimizer state in `float32`.

5. [extended] **Crash-safe manifest.** ☒

1. Build a utility that writes model state, optimizer state, and a JSON metadata file under a fresh step-numbered prefix, then publishes the three by renaming into place one manifest that records each file's name, size, and checksum.
2. Simulate two failures: a crash after the three files are written but before the manifest rename, and the truncation of one file of a published checkpoint. Show that reloading resumes from the previous complete checkpoint in the first case and rejects the corrupt checkpoint in the second.
3. A rename is atomic, so a reader sees either the old manifest or the new one. POSIX does not require the data of the renamed files to reach the disk before the rename does. `os.fsync` forces the buffered writes of a file, or of a directory's entries, to disk. State where the utility must call `os.fsync` so that a power loss leaves either the previous checkpoint or the new one intact, and explain each call.

5.7 GPUs, Devices, and Memory

Since Section 2.4 our `Trainer` has accepted an argument `num_gpus` and ignored it. Nobody was harmed: the models of the earlier chapters were small enough that it hardly mattered where they ran. For larger models it matters a great deal. A current desktop GPU multiplies two 4096×4096 matrices in under 2 milliseconds, where a CPU takes a fifth of a second or more (Section 5.7.2.1 measures both). Unfortunately the speed comes with a constraint: the GPU has its own memory, 24 GB on such a card, and every array it computes with must be there.

Will a given model fit? The model with 8 billion parameters that this chapter keeps returning to needs 16 GB for its weights in bf16, which fits. Training it with Adam in fp32 needs 128 GB by the count of Section 5.2, before a single activation is stored. That does not fit. Whether a model fits on a device, and whether the device computes or waits, is decided by where each array lives and how many bytes each array takes.

We ask three questions in turn. Where does an array live, and how does it move between the CPU and a GPU? When does the CPU wait for the GPU, and how do we time GPU work correctly? What fills GPU memory during training, and how can we trade computation for memory? The answers go into the device-aware `Trainer` that every later chapter uses. Section 5.2 and Section 5.5 counted the bytes per parameter and per number; this section puts those bytes on a device. Training on several GPUs at once follows in Chapter 13. All examples fall back to the CPU when no GPU is present. They then show that the code runs, not how fast it runs on a GPU.

```
import time
from torch.utils.checkpoint import checkpoint
```

```
torch.set_float32_matmul_precision('high')  # allow tf32 in fp32 matmuls
```

5.7.1 Devices and Placement

5.7.1.1 Device Objects

Every array lives on a *device*: the CPU together with main memory, or one GPU together with its own memory. To see which GPUs a machine has, run `nvidia-smi` in a shell; it lists every card, its memory, and the processes using it. Each framework has its own names for devices. The names are soon forgotten; what matters in this part of the section is a single fact, which we reach in a moment: an operation runs on one device, and all its operands must be there.

The CPU is `torch.device('cpu')`. `torch.device('cuda:0')` is the first GPU, `cuda:1` the second, and plain `'cuda'` is `cuda:0`.

We wrap device construction in two functions that the rest of the book uses, and a third counts the GPUs that are available.

```
def cpu():
    """Get the CPU device."""
    return torch.device('cpu')

def gpu(i=0):
    """Get a GPU device."""
    return torch.device(f'cuda:{i}')

def num_gpus():
    """Get the number of available GPUs."""
    return torch.cuda.device_count()
```

Constructing a device object always succeeds, whether or not the hardware exists; the check happens only when data is put on the device.

With the count we write two functions that fall back to the CPU when the requested GPUs do not exist. They let the same code run on a laptop and on a GPU server.

```
def try_gpu(i=0):
    """Return gpu(i) if exists, otherwise return cpu()."""
    if num_gpus() >= i + 1:
        return gpu(i)
    return cpu()

def try_all_gpus():
    """Return all available GPUs, or [cpu(),] if no GPU exists."""
    devices = [gpu(i) for i in range(num_gpus())]
    return devices if devices else [cpu()]

try_gpu(), try_gpu(10), try_all_gpus()
```

```
(device(type='cuda', index=0),
 device(type='cpu'),
```

```
[device(type='cuda', index=0), device(type='cuda', index=1)])
```

Other accelerators, such as the GPUs of Apple-silicon Macs and Google's TPUs, have device names of their own. This book uses the CPU and CUDA GPUs, and the placement rules below carry over to the others.

5.7.1.2 Arrays on Devices

The property `device` tells where an array lives. Where a new array is created by default depends on the framework.

By default, tensors are created on the CPU.

It is better to create data directly on the device where it is used than to create it on the CPU and copy it: the array then never occupies main memory or crosses the bus between CPU and GPU.

To create a tensor on another device, pass the argument `device`.

We put an array `X` on the first GPU and an array `Y` on the second, and add them. On a machine with fewer GPUs, `try_gpu(1)` returns the CPU, and the examples below run without the second device.

```
X = torch.ones(2, 3, device=try_gpu())
Y = torch.rand(2, 3, device=try_gpu(1))
try:
    print(X + Y)
except RuntimeError as e:
    print(str(e).split(', but')[0])
```

```
Expected all tensors to be on the same device
```

The addition fails: PyTorch requires the operands of an operation to be on the same device.

5.7.1.3 Copying Between Devices

A transfer between devices is slow compared with arithmetic on either of them (Section 5.7.2.1 measures how slow), so a framework that copied silently would hide a cost at the place where it occurs. We copy explicitly instead, as in Figure 5.10, and then add.

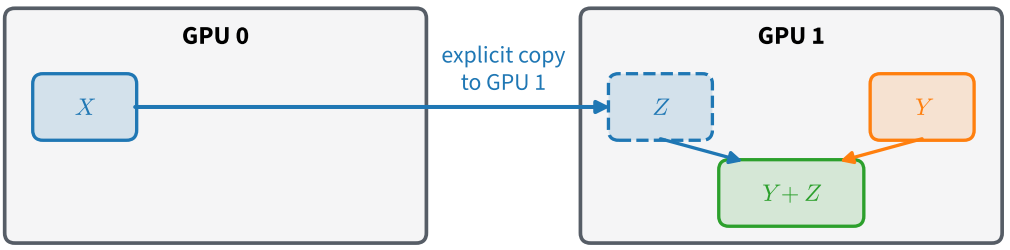


Figure 5.10: X lives on GPU 0 and Y on GPU 1. The copy Z of X on GPU 1 (dashed) lets Y + Z run entirely on GPU 1.

```
Z = X.to(try_gpu(1))
(Y + Z).device
```

```
device(type='cuda', index=1)
```

The sum lives on the second GPU, where both of its operands are. The rule that follows is to move data to the device at the boundary of a computation, once (a move to the device where the data already are costs next to nothing), and to keep the operations inside a training loop on one device.

5.7.1.4 Models on a Device

A model computes with its parameters and its inputs together, so its parameters must live on the device of its data. Every framework places a whole model at once, without our listing its parameters.

A model is a tree of parameter tensors, and `net.to(device)` moves the whole tree in one call.

```
net = nn.Sequential(nn.Linear(3, 1))
net = net.to(device=try_gpu())
net(X)
```

```
tensor([[-1.3715],
        [-1.3715]], device='cuda:0', grad_fn=<AddmmBackward0>)
```

The input and the parameters are on the same device, so the output is computed and stored there too.

Two rules keep a model on its device. Move the model *before* creating the optimizer, so that the optimizer state is created next to the parameters it updates. And when `forward` needs a new tensor, create it on the input's device (`torch.zeros(n, device=X.device)`) rather than on the CPU; state that belongs to the model but is not a parameter goes into a buffer (Section 5.2), which `.to(device)` moves with the parameters.

5.7.2 Asynchronous Execution

5.7.2.1 The Execution Queue

A GPU does not execute operations at the moment Python calls them. Python places each operation in a queue on the device and moves on, while the GPU works through the queue in order. This lets Python prepare the next operation while the GPU computes, but it also means that the time a Python statement takes says nothing about the time its computation takes. We queue 32 products of 4096×4096 matrices and read the clock twice: when Python has issued them, and when the device has finished them.

```
if torch.cuda.is_available():
    A = torch.randn(4096, 4096, device=gpu())
    A @ A
    torch.cuda.synchronize()  # Warm up and wait
    t = time.perf_counter()
    for _ in range(32):
```

```
        B = A @ A
    print(f'time to queue 32 matrix products: {time.perf_counter() - t:.4f} sec')
    torch.cuda.synchronize()
    print(f'time until they all finished:     {time.perf_counter() - t:.4f} sec')
else:
    print('No CUDA device: operations on the CPU run synchronously.')
```

```
time to queue 32 matrix products: 0.0006 sec
time until they all finished:     0.0535 sec
```

CUDA operations are asynchronous, and `torch.cuda.synchronize()` waits until every queued operation on the GPU has finished. Operations on the CPU run synchronously, which is why the cell needs a GPU. The imports cell allows tf32 for fp32 matrix products (Section 5.5), so that the matrix products timed in this section run on the tensor cores.

Python issued the 32 products in a small fraction of the time the device needed to compute them. So when does Python wait at all? Only when it needs a value: converting a result to a Python number or a NumPy array, printing it, or branching on it in an `if`. Each such *synchronization point* blocks Python until the queue up to that value has drained, and then the device sits idle until Python sends more work, as Figure 5.11 shows. A training loop that prints `loss.item()` after every step synchronizes once per step. This is why the steps of the book's `Trainer` return their losses as arrays: `fit_epoch` adds them up on the device in a `d2l.Averager` and converts the sums into Python numbers once per epoch (Section 2.4).

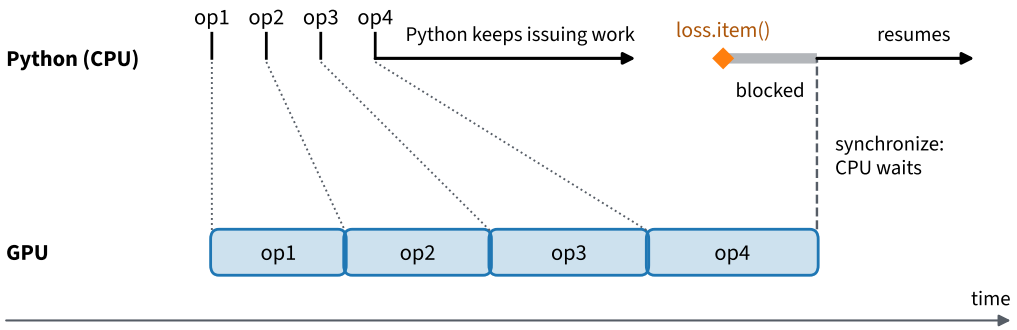


Figure 5.11: Python queues operations op1 through op4 and continues while the GPU runs them in order. loss.item() is a synchronization point: the CPU blocks until the GPU has drained its queue; Python then resumes, and the GPU idles until it receives more work.

Asynchronous execution also dictates how to time GPU code, and a timing that ignores it measures the queue, not the computation. Run the computation once before timing it, so that one-time costs such as memory allocation, library initialization, and compilation are excluded. Wait for the device before starting and before stopping the clock. Time several repetitions and divide, and read the clock with `time.perf_counter`, whose resolution is far finer than a millisecond. The cell above warms up, waits, and reads `perf_counter`, but it reports the total for its 32 products without dividing; the timings later in this section follow all four rules.

`Benchmark` packages these rules for this and later chapters. It calls a function `warmup` times, waits for the device, times `repeats` further calls, waits again, and stores the average time per

call in its attribute `time`.

```
class Benchmark:
    """Time a callable: warmup, then device-synchronized average seconds.
    `devices` lists the GPUs to wait for, by default the current one."""
    def __init__(self, f, warmup=3, repeats=10, desc='time', devices=None):
        def wait():  # CUDA runs asynchronously; the CPU does not
            if torch.cuda.is_available():
                for device in devices or [None]:
                    torch.cuda.synchronize(device)
        self.desc = desc
        for _ in range(warmup):
            f()
        wait()
        t0 = time.perf_counter()
        for _ in range(repeats):
            f()
        wait()
        self.time = (time.perf_counter() - t0) / repeats

    def __repr__(self):
        return f'{self.desc}: {1000 * self.time:.2f} ms/call'
```

We put `Benchmark` to work at once on two claims made so far: that a GPU multiplies matrices far faster than a CPU, and that a transfer between devices is slow compared with arithmetic. We time the product of the queue experiment on the CPU and on the GPU, the same product in a 16-bit format, and the copy of the matrix from main memory to the GPU. On the GPU, the fp32 product runs in tf32 mode on the tensor cores (Section 5.5).

```
if torch.cuda.is_available():
    A_cpu = A.cpu()  # The same matrix in main memory
    A16 = A.bfloat16()
    print(Benchmark(lambda: A_cpu @ A_cpu, warmup=1, repeats=3,
                    desc='product on the CPU, fp32'))
    print(Benchmark(lambda: A @ A, desc='product on the GPU, tf32'))
    print(Benchmark(lambda: A16 @ A16, desc='product on the GPU, bf16'))
    print(Benchmark(lambda: A_cpu.to(gpu()),
                    desc='copy from CPU to the GPU'))
else:
    print('No CUDA device: there is nothing to compare the CPU with.')
```

```
product on the CPU, fp32: 935.38 ms/call
product on the GPU, tf32: 1.67 ms/call
product on the GPU, bf16: 0.91 ms/call
copy from CPU to the GPU: 4.82 ms/call
```

The GPU computes the product in under 2 ms. The CPU takes between a fifth of a second and two seconds, a hundred to a thousand times as long; where in that range it lands depends on how many CPU cores take part. In 16 bits the product takes about half the time that it takes in tf32, which is the speed that Section 5.5 promised and could not show on a CPU.

The last line is the one to remember. Copying the matrix to the GPU takes about three to five times as long as multiplying it there, although the product performs 4096 multiplications

and additions for every number copied. An ordinary copy from main memory first moves the data into *pinned* (page-locked) memory, which the GPU's copy engine can read directly. Data that are pinned already skip that step, and their copy can overlap with computation, as Section 13.2 measures.

5.7.3 GPU Memory

5.7.3.1 Three Views of Memory

How much memory does a program use on a GPU? The answer depends on who is asked. A GPU *allocator* requests memory from the CUDA driver in large blocks and hands out pieces of them, and when an array is freed, it keeps the block for the next array instead of returning it, because requests to the driver are slow. There are therefore three nested views of memory, Figure 5.12: the bytes held by live arrays, the bytes the allocator has reserved, and the bytes the driver has given to the process, which `nvidia-smi` reports.

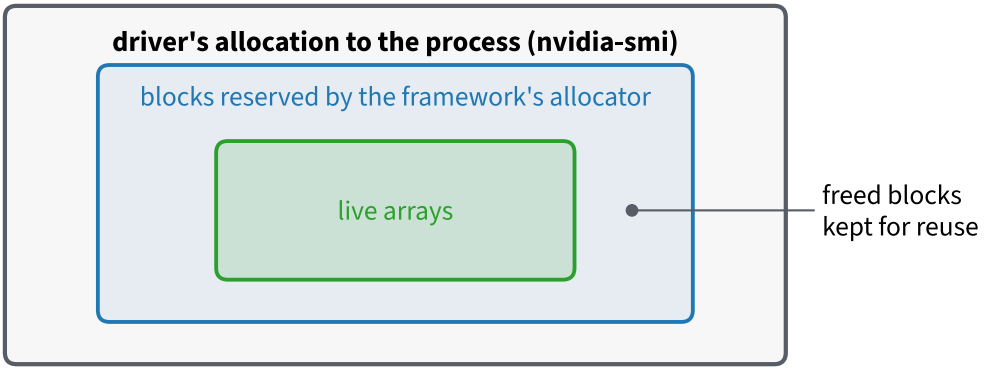


Figure 5.12: The three memory views nest: the driver's allocation to the process, which nvidia-smi reports, contains the allocator's reserved blocks, which contain the live arrays.

`torch.cuda.memory_allocated()` counts the bytes held by live tensors, and `torch.cuda.memory_reserved()` counts those bytes plus the blocks that PyTorch's *caching allocator* keeps for reuse.

We create an array of 1 GiB and delete it again:

```
if torch.cuda.is_available():
    def report(tag):
        print(f'{tag}: {torch.cuda.memory_allocated() / 2**20:7.1f} MiB '
              f'allocated, {torch.cuda.memory_reserved() / 2**20:7.1f} MiB '
              'reserved')
    big = torch.zeros(256, 1024, 1024, device=gpu())  # 1 GiB of float32
    report('after creating big')
    del big
    report('after deleting it ')
else:
    print('No CUDA device: both counters read 0 on the CPU.')
```

```
after creating big:  1192.1 MiB allocated,  1302.0 MiB reserved
after deleting it :   168.1 MiB allocated,  1302.0 MiB reserved
```

After the deletion, `memory_allocated()` falls by a gibibyte while `memory_reserved()` does not move: the block went back to the cache, and `nvidia-smi` still counts it. (Most of what remains allocated is the matrices of the timing cells above: `A` and `B`, 64 MiB each, and the 32 MiB of `A16`.) `torch.cuda.empty_cache()` returns cached blocks to the driver, which helps when another process needs the GPU. Do not call it inside a training loop, since the allocator would then have to request the memory from the driver again at every step.

The memory that limits what fits is the outermost view, since the driver cannot give a process more than the device has. The counts of live bytes are what we use to find out where the memory goes. Of these, the one that decides whether a computation fits is its *peak*, the most memory it holds at any moment.

The allocator tracks this high-water mark: `torch.cuda.reset_peak_memory_stats()` clears it and `torch.cuda.max_memory_allocated()` reads it back. `peak_memory` returns the memory that a call needs beyond what was allocated before it; later chapters use it to compare implementations.

```
def peak_memory(f, *args):
    """Extra peak GPU memory allocated by f(*args), in bytes."""
    torch.cuda.synchronize()
    torch.cuda.reset_peak_memory_stats()
    base = torch.cuda.memory_allocated()
    f(*args)
    torch.cuda.synchronize()
    return torch.cuda.max_memory_allocated() - base
```

5.7.3.2 Memory During Training

For a model with N parameters that trains with Adam in fp32, Section 5.2 counted $16N$ bytes: $4N$ each for the weights and their gradients, and $8N$ for Adam's two running averages. The remaining term, the *activations*, is of a different kind. Backpropagation needs the intermediate results of the forward pass to compute the gradients, so it stores them until the backward pass has used them. Their size grows with the minibatch size and with the width of every layer, while $16N$ does not. Let us predict the terms before we measure them. We take an MLP with three hidden layers of width 4096 on inputs of width 1024, $N \approx 3.8 \times 10^7$ parameters, so that $4N \approx 144$ MiB, at a minibatch size of 4096. The activations that backpropagation keeps are mainly the outputs of the three hidden layers, each a 4096×4096 array of fp32 numbers, 64 MiB apiece: 192 MiB in all, about a third more than the weights.

We read `memory_allocated()` at four points of one training step.

```
if torch.cuda.is_available():
    start = torch.cuda.memory_allocated()
    def allocated():  # Memory in use beyond that at the start of the cell
        return f'{(torch.cuda.memory_allocated() - start) / 2**20:7.1f} MiB'
    net = nn.Sequential(nn.Linear(1024, 4096), nn.ReLU(),
                        nn.Linear(4096, 4096), nn.ReLU(),
                        nn.Linear(4096, 4096), nn.ReLU(),
```

```
                        nn.Linear(4096, 10)).to(gpu())
    optimizer = torch.optim.Adam(net.parameters())
    X = torch.randn(4096, 1024, device=gpu())
    y = torch.randint(0, 10, (4096,), device=gpu())
    print('weights           ', allocated())
    loss = F.cross_entropy(net(X), y)
    print('+ activations     ', allocated())
    loss.backward()
    print('+ gradients       ', allocated())
    optimizer.step()
    print('+ optimizer state ', allocated())
else:
    print('No CUDA device: run this cell on a GPU to see the four readings.')
```

```
weights             160.2 MiB
+ activations       353.4 MiB
+ gradients         313.6 MiB
+ optimizer state   602.0 MiB
```

The first reading is the weights, about $4N$ bytes, plus the input minibatch. The second adds the stored activations, here about a third more than the weights. The third *falls*: `backward` frees the activations as it uses them and leaves gradients of the same size as the weights. The fourth adds about $8N$ at the first `optimizer.step()`, when Adam creates its two running averages. From then on every step rises to the second reading and falls back to the third; weights, gradients, and optimizer state stay.

The minibatch size is the first memory control to try, because it scales the activations without changing the model. If memory instead grows from one step to the next, something keeps the arrays of earlier steps alive. The usual suspect is a Python list of losses: a loss that still carries its computation graph keeps the activations of its step alive with it, so detach the loss before storing it.

5.7.3.3 Activation Checkpointing

The activations have a second control. Backpropagation stores every intermediate result only to read it once, on the way back. *Activation checkpointing* (Chen et al. 2016) stores less: it keeps 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. (The name has nothing to do with the checkpoint files of Section 5.6: here a checkpoint is an activation kept in memory.) For a stack of K equally sized layers cut into segments of s layers, the backward pass holds the K/s segment inputs plus the activations of one segment, about $K/s + s$ layer activations instead of K. The sum is smallest at $s = \sqrt{K}$, where stored activations fall from $O(K)$ to $O(\sqrt{K})$. The saving costs computation, namely a second forward pass through every segment, Figure 5.13. How much longer does a step take? A backward pass costs about twice a forward pass: for a linear layer it computes two matrix products, one for the gradient of the weights and one for the gradient of the input, where the forward pass computes one. A step then costs about four forward passes instead of three, a third more. This is a rather crude count, and the exact overhead depends on the layers and the hardware.

Checkpointing suits deep stacks of identical blocks, such as the residual stack of Section 5.1 and the transformers of later chapters. We use the residual block of Section 5.1,

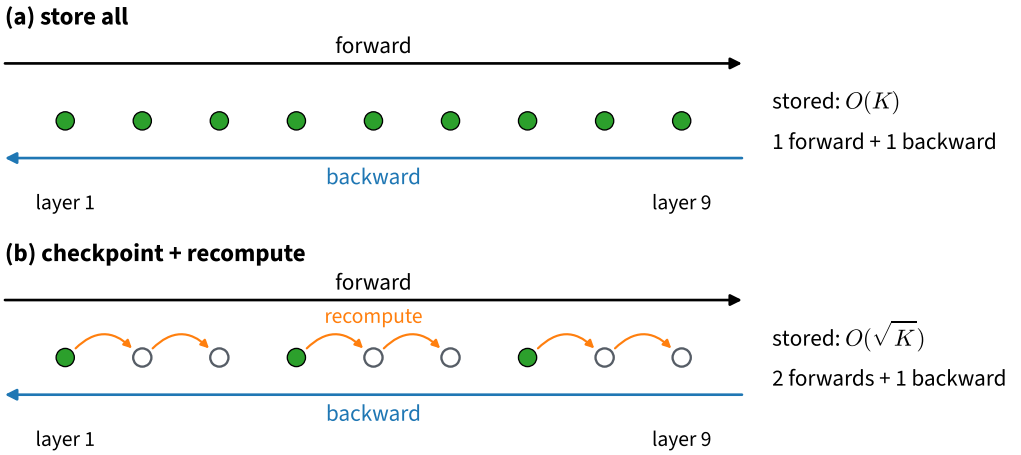


Figure 5.13: (a) Standard backpropagation stores the activations of every layer. (b) Checkpointing stores only the inputs of the segments (filled) and recomputes the activations inside each segment (open) during the backward pass. With K layers in segments of $\sqrt{K}$ layers, stored activations fall from $O(K)$ to $O(\sqrt{K})$ at the cost of one extra forward pass.

$\mathbf{X} + \text{Linear}(\text{ReLU}(\text{Linear}(\mathbf{X})))$, with its weights drawn from $\mathcal{N}(0, 0.01^2)$ and zero biases in every framework, so that all tabs build the same model. A function runs a stack of blocks in segments, with or without checkpointing.

```
class ResidualBlock(nn.Module):
    def __init__(self, num_hiddens):
        super().__init__()
        self.body = nn.Sequential(
            nn.Linear(num_hiddens, num_hiddens), nn.ReLU(),
            nn.Linear(num_hiddens, num_hiddens))
        for layer in self.body[0], self.body[2]:
            nn.init.normal_(layer.weight, std=0.01)
            nn.init.zeros_(layer.bias)

    def forward(self, X):
        return X + self.body(X)

def run_stack(blocks, X, use_checkpoint=False, segment_size=4):
    for start in range(0, len(blocks), segment_size):
        stop = min(start + segment_size, len(blocks))
        # Bind this segment's range now: the backward pass calls segment
        # again after the loop has moved on
        def segment(X, start=start, stop=stop):
            for i in range(start, stop):
                X = blocks[i](X)
            return X
        X = checkpoint(segment, X, use_reentrant=False) \
            if use_checkpoint else segment(X)
```

```
        return X
```

`checkpoint(segment, X)` runs a segment without storing its intermediate results and keeps what it needs to rerun it. The recomputation must reproduce the original forward pass, so `checkpoint` also saves and restores the state of the random number generator, which keeps segments with dropout correct.

Recomputation must not change the result. We check that the gradients through a stack of eight blocks, run as two checkpointed segments, equal the ordinary ones.

```
blocks = nn.Sequential(*[ResidualBlock(64) for _ in range(8)])
X = torch.randn(32, 64)

def stack_grads(use_checkpoint):
    blocks.zero_grad(set_to_none=True)
    run_stack(blocks, X, use_checkpoint).sum().backward()
    return [p.grad.clone() for p in blocks.parameters()]

for g, g_ckpt in zip(stack_grads(False), stack_grads(True)):
    assert torch.allclose(g, g_ckpt)
print('checkpointed gradients match the ordinary ones')
```

```
checkpointed gradients match the ordinary ones
```

The check runs on the CPU too. The memory effect needs a GPU and a stack deep enough for a clear difference: sixteen blocks of width 1024 at a minibatch size of 8192, in segments of four blocks. We report the *step memory*, the peak memory of a step beyond the parameters and the input it starts from, and the time per step that `Benchmark` measures.

```
if torch.cuda.is_available():
    blocks = nn.Sequential(*[ResidualBlock(1024) for _ in range(16)]).to(gpu())
    X = torch.randn(8192, 1024, device=gpu())
    def step(use_checkpoint):
        blocks.zero_grad(set_to_none=True)
        run_stack(blocks, X, use_checkpoint).sum().backward()
    for use_checkpoint in (False, True):
        timing = Benchmark(lambda: step(use_checkpoint))
        blocks.zero_grad(set_to_none=True)  # Gradients count as step memory
        peak = peak_memory(step, use_checkpoint)
        print(f'checkpointing={use_checkpoint!s:5}  step memory '
              f'{peak / 2**20:5.0f} MiB, {timing.time * 1e3:5.1f} ms per step')
else:
    print('No CUDA device: the peak-memory comparison needs a GPU.')
```

```
checkpointing=False  step memory  1096 MiB,  25.2 ms per step
checkpointing=True   step memory   452 MiB,  32.5 ms per step
```

Without checkpointing, the step holds the activations of all sixteen blocks. With segments of four blocks, it holds the four segment inputs plus the activations of one segment, and its memory falls from about 1.1 GiB to between 0.4 and 0.5 GiB. The step takes about a quarter longer, a little less than the third that our crude count of forward passes predicted. This

trade lets a model train when its activations would otherwise exceed the memory of the device, and it is common in the training of large transformers.

5.7.4 Device-Aware Trainer

The `Trainer` can at last put its argument `num_gpus` to use. It needs nothing beyond the rules of this section: the parameters are on the device before training starts, and every minibatch is moved there at the start of its step.

`prepare_model` moves the model before the `Trainer` creates the optimizer, and `prepare_batch` moves every minibatch.

```
@d2l.add_to_class(d2l.Trainer)
def __init__(self, max_epochs, num_gpus=0, gradient_clip_val=0):
    self.save_hyperparameters()
    self.gpus = [d2l.gpu(i) for i in range(min(num_gpus, d2l.num_gpus()))]
    self.board = d2l.ProgressBoard(xlabel='epoch', ylabel='loss')

@d2l.add_to_class(d2l.Trainer)
def prepare_batch(self, batch):
    if self.gpus:
        batch = [d2l.to(a, self.gpus[0]) for a in batch]
    return batch

@d2l.add_to_class(d2l.Trainer)
def prepare_model(self, model):
    self.model = model
    if self.gpus:
        model.to(self.gpus[0])
        # Pick convolution algorithms by timing, unless determinism is wanted
        torch.backends.cudnn.benchmark = not (
            torch.are_deterministic_algorithms_enabled()
            or torch.backends.cudnn.deterministic)
    # Lazy parameters get their shapes from a dry pass on the device, then
    # the model's weight_init, if it has one, before the optimizer exists
    if any(isinstance(p, nn.parameter.UninitializedParameter)
           for p in model.parameters()):
        loader = (self.train_dataloader if self.train_dataloader is not None
                  else self.val_dataloader)  # `evaluate` has no training data
        batch = self.prepare_batch(next(iter(loader)))
        model.apply_init(batch[:-1], getattr(model, 'weight_init', None))
```

The expression `min(num_gpus, d2l.num_gpus())` makes the request an upper limit rather than a demand. On a machine without a GPU, `self.gpus` is empty, nothing is moved to a GPU, and training runs on the CPU. Every `Trainer(num_gpus=1)` in the rest of the book relies on this fallback.

Does a GPU pay off in training? The product of two large matrices ran more than a hundred times faster on the GPU than on the CPU, but a training step is more than one large product. We answer for a classifier built from the residual blocks of Section 5.7.3.3: a hidden layer of width 256, two residual blocks, and 10 outputs, with all weights drawn from $\mathcal{N}(0, 0.01^2)$ and zero biases, as for the MLPs of earlier chapters.

```
class ResMLPClassifier(d2l.Classifier):
    def __init__(self, num_hiddens=256, num_blocks=2, lr=0.1):
        super().__init__()
        self.save_hyperparameters()
        self.net = nn.Sequential(
            nn.Flatten(), nn.Linear(784, num_hiddens), nn.ReLU(),
            *[ResidualBlock(num_hiddens) for _ in range(num_blocks)],
            nn.Linear(num_hiddens, 10))
        for layer in self.net[1], self.net[-1]:  # The blocks set their own
            nn.init.normal_(layer.weight, std=0.01)
            nn.init.zeros_(layer.bias)
```

We train the model on Fashion-MNIST with `num_gpus=0` and with `num_gpus=1`, and time one epoch after one epoch of warm-up.

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

def epoch_time(num_gpus):
    model = ResMLPClassifier()
    trainer = d2l.Trainer(max_epochs=1, num_gpus=num_gpus)
    trainer.board.display = False
    trainer.fit(model, data)  # The first epoch warms up
    return Benchmark(trainer.fit_epoch, warmup=0, repeats=1).time

for gpus in (0, 1):
    print(f'num_gpus={gpus}: {epoch_time(num_gpus=gpus):.1f} sec per epoch')
```

```
num_gpus=0: 2.2 sec per epoch
num_gpus=1: 1.9 sec per epoch
```

This is a sobering result. On the GPU an epoch runs at most a few times as fast as on the CPU, far from the factor of more than a hundred that the GPU gained on the large matrix product. A step of this model multiplies matrices with at most 784×256 entries, too little work to keep the GPU busy, and much of an epoch goes into loading the data and launching the many small operations of each step. A GPU pays off when large matrix products dominate the work, as they do in the larger models of the chapters to come.

The memory of this run follows the accounting above: weights, gradients, and the state of the optimizer, which for SGD is nothing, plus the activations of one minibatch. It is a small model, and it fits with room to spare. And the 8-billion-parameter model of the opening? A smaller minibatch and activation checkpointing shrink its activations, but its 128 GB of weights, gradients, and optimizer state are needed before the first activation is stored, and no setting of this section touches them. Note that the `Trainer` would not have warned us: it cannot tell whether a model fits before it runs out of memory. That estimate we still make by hand, from the bytes per parameter and the size of the activations. Training on more than one device is taken up in Chapter 13.

5.7.5 Summary

Every array and every parameter lives on a device, and an operation combines arrays on one device. Copies between devices are explicit and slow compared with arithmetic, so they belong at the boundary of a computation, not inside the training loop. A GPU executes

its queue asynchronously, so Python waits only at synchronization points, such as converting a result to a Python number, and correct timing needs a warm-up, explicit waits, and repetitions. GPU memory during training holds the weights, their gradients, and the optimizer state, which depend only on the model, plus activations that grow with the minibatch size. Activation checkpointing stores only the inputs of segments and recomputes the rest, which lowers the peak memory at the cost of one extra forward pass. The device-aware `Trainer` moves the model once and every minibatch per step, and it falls back to the CPU when no GPU is present.

5.7.6 Exercises

1. code **Peak memory.** Take a four-layer MLP with widths 1024, 4096, 4096, 4096 and 10 outputs, trained with Adam, as in Section 5.7.3.2. Using the accounting of that subsection, predict the peak memory of one training step at batch sizes 64, 256, and 1024, counting weights, gradients, optimizer state, and the activations the backward pass keeps. Then measure the peak on a GPU. Where does the prediction break down, and what did it omit?

2. **Checkpointing arithmetic.** Work out the schedule of Figure 5.13 on paper for a stack of sixteen `ResidualBlock`s of width 1024 at batch size 8192.

    1. Name the arrays inside one block that the backward pass must keep, and count their entries.
    2. Compute the activation memory held at the peak of the backward pass without checkpointing and with segments of one, two, four, and eight blocks. State how much forward computation each schedule repeats.
    3. Which segment size would you choose here, and how does it relate to the rule of segments of about $\sqrt{K}$ layers in Section 5.7.3.3?

3. code **Asynchronous dispatch.** ▯ Use the timing rules of Section 5.7.2.1 for both parts.

    1. For n from 256 to 8192, time a round trip of an $n \times n$ matrix from CPU to GPU and back, and time one $n \times n$ matrix product on the device. Find the n at which the two times are equal, extrapolating from how each time grows with n if they do not cross in this range. Predict this crossover from your GPU's published host-to-device bandwidth and matrix-product throughput, and explain any gap between prediction and measurement.
    2. If you have two GPUs, time 1000 matrix products of two 4096×4096 matrices executed on one GPU, then 500 on each of two GPUs issued from the same loop. Measure the scaling, explain how asynchronous dispatch lets Python issue work to both devices, and identify sources of deviation from linear scaling. Guard the experiment with `num_gpus() >= 2`.

4. code **Precision crossover.** ▯ Adapt the `train` function of Section 5.5 to a GPU: place the network and every batch on the device, so that the 16-bit computation runs there, and let the caller choose the hidden width of the MLP and the batch size. Time ten steps of its fp32 run and of its mixed-precision run with `Benchmark`, after one untimed step.

    1. For hidden widths 256 and 4096, sweep the batch size from 8 to 4096 in powers of two and plot the mixed-precision speedup over fp32 against the batch size, one curve per width.

2. Below which batch size, if any, is mixed precision slower than fp32 at each width? Explain the reversal and why the crossover moves with the width.

Adapted from the PyTorch recipe Automatic Mixed Precision.

5. code **Synchronization points.** ☒ Consider the training run of `ResMLPClassifier` in Section 5.7.4 on a GPU.
   1. List every synchronization point in one training step, reading `Trainer.fit_epoch`, `Trainer.prepare_batch`, and `Averager` as well as the model code. Classify each as unavoidable for a correct loop or as an avoidable instrumentation artifact, and justify each classification.
   2. Suppose the loop reads the loss into a Python float after every step. Predict from your list how much this readout slows down the default `ResMLPClassifier` and `ResMLPClassifier(num_hiddens=2048, num_blocks=8)`. Then time one epoch of each model with and without the readout, and explain the four timings.

Adapted from the PyTorch documentation CUDA semantics.

6. code **Activation checkpointing.** Revisit the comparison of Section 5.7.3.3.
   1. Increase the batch size until the run without checkpointing raises an out-of-memory error. How much further can the checkpointed run go before it does too? Explain the ratio from what each variant stores.
   2. Run `run_stack` with `segment_size` equal to 1, 4, and the number of blocks, and once without checkpointing. Plot step memory against step time for the four configurations. Which segment size would you choose here, and why?

5.8 Reproducibility and Inspection

Suppose we change one line of a training script and the validation accuracy of a Fashion-MNIST classifier rises by 0.3 points. Before we credit the change, we need to know how much two runs of the unchanged script differ. For softmax regression we measure the answer in Section 5.8.1.5: five runs that differ only in their seed end one to four points apart, because each run draws different initial weights and visits the examples in a different order. A gain of 0.3 points is well inside that range.

A run that fails poses a second problem of the same kind. If the loss turns into NaN after hours of training, we want to repeat the run exactly up to that point and then look inside the model to see which layer produced the first non-finite number.

This section first makes runs repeatable. Every random draw comes from a generator whose sequence of numbers is fixed by a seed, so seeding the generators fixes the draws. That is the easy part. Repeating a run bit for bit also requires that the arithmetic adds numbers in a fixed order, which a GPU does not promise. The section then looks inside running models with *hooks*, functions that observe each layer of an unmodified model as it computes. The two halves serve one purpose: a measurement taken inside a model can be compared across two versions of the code only if the runs differ in nothing but that code. On a first reading, the subsections on generator objects and on loader workers can be skimmed; they matter once a data pipeline draws random numbers of its own.

```
from torch.utils.data import DataLoader, Dataset
```

5.8.1 Seeds and Random Number Generators

Randomness enters a training run in more places than one might list at first: initialization draws every weight (Section 5.3), dropout draws a new mask at every step, the data loader shuffles the examples differently in every epoch, data augmentation draws crops and flips, and parallel loader workers do some of this work in separate processes. Each draw comes from a *pseudorandom number generator*, a deterministic algorithm whose sequence of outputs is fixed by its starting state, the *seed*. Seeding every generator that a run consults therefore fixes every draw.

5.8.1.1 Global Seeds

A framework's random functions either consult a global generator or take their state as an argument. Which of the two a framework does decides how a run is seeded. To see a seed at work, the function below trains a small network for five steps on random data and draws everything, weights and data, from the one source that its seed fixes.

`torch.manual_seed(k)` seeds PyTorch's global generator on the CPU and on every GPU. It does not seed Python's `random` module or NumPy's generators, which carry seeds of their own.

```
def train_once(seed):
    torch.manual_seed(seed)  # seeds the CPU and every GPU
    net = nn.Sequential(nn.Linear(20, 32), nn.ReLU(), nn.Linear(32, 1))
    opt = torch.optim.SGD(net.parameters(), lr=0.1)
    X, y = torch.randn(128, 20), torch.randn(128, 1)
    for _ in range(5):
        opt.zero_grad()
        loss = ((net(X) - y) ** 2).mean()
        loss.backward()
        opt.step()
    return loss.item(), net[0].weight.detach().clone()

loss_a, w_a = train_once(seed=0)
loss_b, w_b = train_once(seed=0)
loss_c, _ = train_once(seed=1)
print('same seed, identical loss and weights:',
      loss_a == loss_b, torch.equal(w_a, w_b))
print('different seed:', loss_a, 'vs', loss_c)
```

```
same seed, identical loss and weights: True True
different seed: 0.8643209338188171 vs 0.9312677383422852
```

The two runs with seed 0 agree *bitwise*, down to the last bit of the loss and of every weight: same initial weights, same data, same gradients, same operations in the same order. (The order is the one thing a seed does not control; we return to it in Section 5.8.2.) The run with seed 1 ends with a different loss.

5.8.1.2 Generator Objects

A single global generator couples all the code that draws from it. Inserting one extra random call, a new layer's initialization or a stray draw while debugging, shifts every number drawn after it, so the data split, the initialization, and the dropout masks all change at once. A *generator object* is a private stream with its own seed, which unrelated draws cannot shift.

A `torch.Generator` is such a stream. Here a random permutation, say a split of a dataset into training and validation examples, stays fixed no matter what else draws from the global generator in between:

```
g = torch.Generator().manual_seed(42)
split = torch.randperm(10, generator=g)
_ = torch.randn(1000)  # unrelated draws from the global generator
split_again = torch.randperm(10, generator=g.manual_seed(42))
print(torch.equal(split, split_again), split)
```

```
True tensor([2, 6, 1, 8, 4, 5, 0, 9, 3, 7])
```

Most sampling functions accept `generator=`, and so does `DataLoader`, where it sets the shuffle order. We use that next.

5.8.1.3 Data Loader Workers

Data loaders often prepare minibatches in parallel *worker* processes. A worker that draws random numbers, for data augmentation for instance, needs a stream of its own. Two failures are common, and neither raises an error. If every worker starts as a copy of the main process, as with the *fork* start method, the workers inherit the same generator state and apply the same "random" augmentations. If each worker seeds itself from the operating system's entropy, no two runs agree.

`DataLoader` gives every worker its own PyTorch seed, derived from a base seed and the worker's index, and seeds Python's `random` module and NumPy's global generator in each worker from it. It cannot reach a generator object that the dataset holds: every worker receives a copy of the dataset, the generator's state included. Such generators need a *worker initializer*, a function that the loader calls once in each new worker; `get_worker_info()` gives it the worker's seed and its copy of the dataset. The loader's `generator` sets the base seed (and the shuffle order), so one generator fixes the whole input pipeline. Below, two workers read a dataset of eight examples, each a draw from the dataset's own generator, first without and then with an initializer. The cell asks for the fork start method, which Linux and macOS provide and Windows does not:

```
class NoisyData(Dataset):
    """Eight examples whose values are random draws, as in augmentation."""
    def __init__(self):
        self.rng = np.random.default_rng()  # the dataset's own generator

    def __len__(self):
        return 8

    def __getitem__(self, i):
        return self.rng.random()

def seed_worker(worker_id):
    info = torch.utils.data.get_worker_info()
    info.dataset.rng = np.random.default_rng(info.seed)  # this worker's seed

for init_fn in (None, seed_worker):
```

```
    loader = DataLoader(NoisyData(), batch_size=4, num_workers=2,
                        worker_init_fn=init_fn,
                        multiprocessing_context='fork',  # copies of this one
                        generator=torch.Generator().manual_seed(0))
    print([[round(x, 3) for x in batch.tolist()] for batch in loader])
```

```
[[0.117, 0.761, 0.904, 0.053], [0.117, 0.761, 0.904, 0.053]]
[[0.777, 0.641, 0.352, 0.467], [0.653, 0.267, 0.932, 0.823]]
```

Without the initializer, the two batches hold the same four numbers: both workers drew from copies of one generator. With it, the numbers differ, and since the worker seeds derive from the loader's generator, the second line is the same in every run. The first line changes from run to run, because the dataset's generator seeded itself from the operating system.

5.8.1.4 Randomness as a Value

The worker failures above arise because the state of a generator is copied or shared without appearing in the program's data flow. JAX makes randomness a value instead: every random function takes a key, the same key gives the same numbers, and independent streams come from splitting a key. A key can still be reused by mistake, but the reuse is visible in the code. The cost is that every function that draws random numbers must accept a key and pass split keys on. A global seed is the convenient form of the same contract: one implicit key that all code shares. Generator objects recover the explicit form where it matters.

PyTorch offers both forms: the global generator, and `torch.Generator` objects passed to the functions that should use them.

5.8.1.5 The Spread across Seeds

We can now measure the spread that the opening asked about. The function below trains the softmax regression of Section 3.4 on Fashion-MNIST for three epochs on a GPU, seeds every generator that the run consults, and returns the final validation accuracy and the trained weights. We run it with the seeds 0 to 4.

`torch.manual_seed` fixes the initialization, the loader's shuffle order, and the seeds of its worker processes.

```
def train_classifier(seed, data):
    torch.manual_seed(seed)  # initialization, shuffle order, worker seeds
    model = d2l.SoftmaxRegression(num_outputs=10, lr=0.1)
    trainer = d2l.Trainer(max_epochs=3, num_gpus=1)
    trainer.board.display = False
    trainer.fit(model, data)
    return trainer.history['val_acc'][-1], model.net[1].weight.detach().cpu()

data = d2l.FashionMNIST(batch_size=256)
runs = [train_classifier(seed, data) for seed in range(5)]
accs = 100 * np.array([acc for acc, _ in runs])
print('validation accuracy (%):', ' '.join(f'{a:.1f}' for a in accs))
print(f'mean {accs.mean():.1f}, max - min {np.ptp(accs):.1f} points')
```

```
validation accuracy (%): 79.8 81.8 81.9 78.3 82.1
mean 80.8, max - min 3.7 points
```

The five runs differ only in their seeds, yet their final accuracies lie one to four points apart, around 81%. A gain of 0.3 points, measured on one run of each variant, is well inside that spread. To credit a change we compare the mean over several seeds per variant, together with the spread.

The seed is all that separated these runs. To confirm it, we repeat the run with seed 0 and compare it with the first one, accuracy and every weight:

```
acc, w = train_classifier(0, data)
print('same accuracy:', acc == runs[0][0],
      '| same weights:', torch.equal(w, runs[0][1]),
      f'| largest difference {(w - runs[0][1]).abs().max().item():.2g}')
```

```
same accuracy: True | same weights: True | largest difference 0
```

The rerun reproduces the accuracy of the first run exactly. Whether every weight agrees as well depends on the order in which the GPU adds numbers, which a seed does not fix (Section 5.8.2).

Here every weight agrees bitwise: the operations of this model add in a fixed order on the GPU.

5.8.2 Deterministic Arithmetic

Seeding fixes which numbers the program draws. It does not fix how the arithmetic combines them. Floating-point addition is not associative: every sum is rounded to the nearest representable number, so adding the same numbers in a different order can give a different result.

Three float32 numbers suffice to show it. Near 10^8, adjacent float32 numbers are 8 apart (Section 5.5), so $-10^8 + 1$ rounds back to -10^8:

```
a, b, c = [torch.tensor(v, dtype=torch.float32) for v in (1e8, -1e8, 1.0)]
print(float((a + b) + c), float(a + (b + c)))  # same numbers, regrouped
```

```
1.0 0.0
```

Regrouping changes the result from 1 to 0. In a long sum the effect is smaller but of the same kind: adding a million random numbers forward and backward usually gives results that differ in their last bits.

On a CPU, the order in which one operation adds its numbers is usually fixed for a given number of threads, so a seeded run repeats. On a GPU, many threads add partial sums at the same time, and some operations combine them with *atomic additions*, which commit each partial sum in whatever order the threads happen to finish. Examples are the gradients of an embedding lookup or of a scatter operation, some convolution gradients, and some reductions. Two seeded runs on the same machine can then differ in the last bits of their weights, and over many training steps such differences grow until the loss curves differ visibly. The remedy is a switch that replaces such operations with deterministic ones, which add in a fixed order and are often slower. The cell below shows both. A *scatter-add* sums a

million random numbers into ten bins, each number into the bin its index names; we run it twice on a GPU and compare the two results bit by bit.

`torch.use_deterministic_algorithms(True)` is the switch. Operations that have a deterministic implementation use it; operations without one raise a `RuntimeError` instead of returning nondeterministic results. The cell runs the scatter-add `index_add_` with the switch off and then on:

```
if torch.cuda.is_available():
    src = torch.randn(2**20, device='cuda')  # a million numbers
    idx = torch.randint(0, 10, (2**20,), device='cuda')  # their bins
    for deterministic in (False, True):
        torch.use_deterministic_algorithms(deterministic)
        a, b = [torch.zeros(10, device='cuda').index_add_(0, idx, src)
                for _ in range(2)]
        print(f'deterministic {deterministic!s:5}: bitwise equal '
              f'{torch.equal(a, b)}, largest difference '
              f'{(a - b).abs().max().item():.2g}')
    torch.use_deterministic_algorithms(False)
else:
    print('no GPU: on a CPU this sum adds in a fixed order')
```

```
deterministic False: bitwise equal False, largest difference 0.0038
deterministic True : bitwise equal True, largest difference 0
```

The two scatter-adds without the switch differ in their last bits, by a few thousandths on bin sums of a few hundred; the deterministic computation agrees with itself bitwise.

Bitwise agreement holds only for one machine, one library version, and one configuration of these switches; no framework promises it across releases, across hardware, or between CPU and GPU. Bitwise reproducibility is therefore a tool for debugging: it lets us find the step at which two runs first differ. The scientific goal is weaker and more robust. A conclusion, such as "the change raises accuracy", should hold across seeds, so we report the mean and the spread over several runs rather than one run, as for the five seeds of Section 5.8.1.5 (about 81%, with one to four points between the best and the worst run). As with the number formats of Section 5.5, a conclusion that changes with the seed or with the last bits of the arithmetic needs more evidence.

5.8.3 Model Inspection with Hooks

A repeatable run tells us that a problem will recur, not where it arises. To find out, we observe what each layer computes during the run. Editing the `forward` method of every layer to print or store its output is tedious, and impossible for a model that came from a library or a checkpoint whose code we do not want to change. A *hook* is a function that the framework calls at a fixed point of each layer's computation, from outside the layer's code.

Hooks run in the `__call__` wrapper around `forward` (Section 5.1). `module.register_forward_hook(f)` arranges for `f(module, inputs, output)` to run after every forward pass of the module, and `register_forward_pre_hook` before it; Figure 5.14 shows where they run. A forward hook that returns a value replaces the module's output.

As a test model we use a stack of eight residual blocks of width 64 between an input layer with 20 inputs and an output layer with 10 outputs. The block is the one of Section 5.1: it

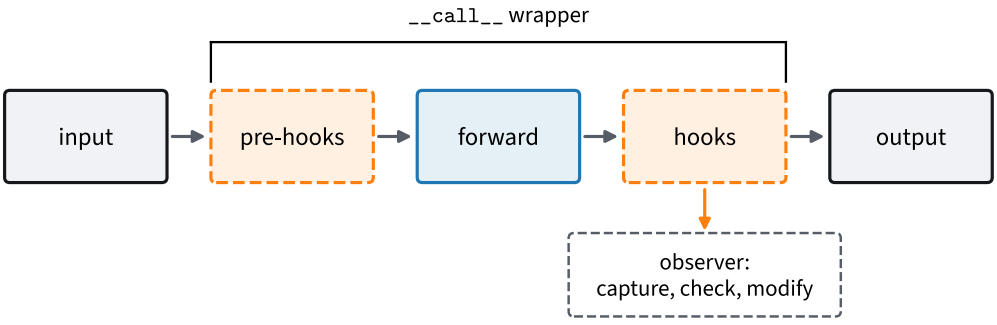


Figure 5.14: The `__call__` wrapper as a pipeline: input flows through pre-hooks, then forward, then hooks, to the output, with the two hook stages dashed and orange against forward's solid blue, and a side arrow from the hooks stage to an observer that can capture, check, or modify.

adds the output of its *body*, a linear layer, a ReLU, and a second linear layer, to its input, $\mathbf{h} = \mathbf{x} + \text{body}(\mathbf{x})$. The weights start from Xavier initialization (Section 4.4.2.1) and the biases at zero. We do not seed this model. Its weights are drawn afresh in every run, which is why the numbers below are quoted as ranges.

The seeded runs above left PyTorch's global generator at a fixed state, which would fix this model's weights too, so the cell first reseeds the generator from the operating system's entropy with `torch.seed()`.

```
torch.seed()  # a fresh seed from the operating system

class ResidualBlock(nn.Module):
    def __init__(self, num_hiddens):
        super().__init__()
        self.body = nn.Sequential(nn.Linear(num_hiddens, num_hiddens),
                                  nn.ReLU(),
                                  nn.Linear(num_hiddens, num_hiddens))

    def forward(self, X):
        return X + self.body(X)

net = nn.Sequential(nn.Linear(20, 64),
                    *[ResidualBlock(64) for _ in range(8)],
                    nn.Linear(64, 10))
for m in net.modules():
    if isinstance(m, nn.Linear):  # Xavier weights, zero biases
        nn.init.xavier_uniform_(m.weight)
        nn.init.zeros_(m.bias)
```

5.8.3.1 Activation Statistics

The initialization experiments of Section 5.3 measured the standard deviation of the activations at every depth. With hooks, the same measurement takes a few lines on an unmodi-

fied model. We record the standard deviation of each layer's output on a minibatch of 256 standard normal inputs.

```
stats, handles = [], []

def record(module, inputs, output):
    stats.append((type(module).__name__, output.detach().std().item()))

for m in net:
    handles.append(m.register_forward_hook(record))
net(torch.randn(256, 20))
for h in handles:
    h.remove()
for name, std in stats:
    print(f'{name:15s} std {std:.2f}')
```

```
Linear          std 0.69
ResidualBlock   std 0.84
ResidualBlock   std 1.14
ResidualBlock   std 1.42
ResidualBlock   std 1.87
ResidualBlock   std 2.16
ResidualBlock   std 2.74
ResidualBlock   std 3.37
ResidualBlock   std 3.75
Linear          std 4.76
```

The standard deviation grows from block to block, from about 0.7 after the input layer to roughly 3 to 4 after the eighth block. Both numbers can be predicted. Xavier initialization gives the input layer, with 20 inputs and 64 outputs, weights of variance $2/84$, so its output has variance $20 \cdot 2/84 \approx 0.48$ and standard deviation 0.69. Each block then adds its body's output to its input, and the two are nearly uncorrelated, so their variances add. The body's output has about half the variance of its input: the first linear layer keeps the variance, the ReLU halves the mean square, and the second linear layer keeps it again. The variance therefore grows by a factor of about 1.5 per block, and the standard deviation by about $\sqrt{1.5} \approx 1.22$. Eight blocks should multiply 0.69 by $1.5^4 \approx 5$, which gives 3.5, and the table agrees to within the scatter of one random draw. This is the calculation of Section 5.3 with a factor of 1.5 where He initialization gave 2, and it is the reason why the bodies of deep residual networks are often initialized with smaller weights.

Two rules keep hooks safe. First, *detach before storing*: the hook above stores `output.detach().std()`. Storing `output` itself would keep the autograd graph of every forward pass in memory until `stats` is cleared, a leak that grows with every minibatch. Second, *keep the handle and remove it*: a hook stays registered for the module's lifetime unless we call `handle.remove()`, slowing every later forward pass and keeping alive any tensors it stores.

5.8.3.2 A NaN Finder

When a loss becomes NaN at step 40,000, the useful question is which layer produced the first non-finite value. A hook on every layer that checks its output for infinities and NaNs answers it in one forward pass: the first hook to fire names the layer, because hooks fire in the order the layers compute. To test this, we put a NaN into one weight of the third residual

block, the module at index 3 of the stack:

```
def make_finite_check(name):
    def check(module, inputs, output):
        if not torch.isfinite(output).all():
            raise RuntimeError(f'first non-finite output in {name}')
    return check

handles = [m.register_forward_hook(make_finite_check(name))
           for name, m in net.named_modules()
           if len(list(m.children())) == 0]  # leaf modules only
saved = net[3].body[0].weight[0, 0].item()
with torch.no_grad():
    net[3].body[0].weight[0, 0] = float('nan')  # break one layer
try:
    net(torch.randn(2, 20))
except RuntimeError as e:
    print(e)
for h in handles:
    h.remove()
with torch.no_grad():
    net[3].body[0].weight[0, 0] = saved  # restore the weight
```

```
first non-finite output in 3.body.0
```

The report names `3.body.0`, the layer that holds the NaN, without a single print statement in the model.

A planted NaN is the easy case, since we knew where to look. In a real run the finder names the first symptom, and the cause lies at or before it: activations may grow over several layers before one of them overflows.

5.8.3.3 Gradient Inspection

Gradients can be inspected in the same way. The forward signal grew with depth. Does the backward signal do the same? We record the norm of the gradient with respect to the output of each residual block, and the gradient norms of the first linear layer of the third residual block. As a loss we take the mean squared output of the stack on a minibatch of 256 standard normal inputs.

`module.register_full_backward_hook(f)` runs `f(module, grad_input, grad_output)` during the backward pass, and `grad_output[0]` is the gradient with respect to the module's output. Like a forward hook, a backward hook returns a handle to remove and should detach anything it stores. The gradients of the parameters need no hook: after the backward pass they are available as `p.grad`.

```
X = torch.randn(256, 20)
grad_norms, handles = [], []
def record(module, grad_input, grad_output):
    grad_norms.append(grad_output[0].norm().item())  # store a number
for block in net[1:-1]:
    handles.append(block.register_full_backward_hook(record))
```

```
net.zero_grad()
(net(X) ** 2).mean().backward()
for h in handles:
    h.remove()
# The hooks fire in backward order, last block first
print('block output gradients:',
      ' '.join(f'{g:.3g}' for g in reversed(grad_norms)))
layer = net[3].body[0]  # block 3, first layer
print(f'weight {layer.weight.grad.norm().item():.3g}, '
      f'bias {layer.bias.grad.norm().item():.3g}')
```

```
block output gradients: 1.46 1.14 0.914 0.715 0.594 0.448 0.347 0.271
weight 26.8, bias 4.87
```

Read from the last block back to the first, the gradient norm grows by a factor of roughly 1.2 to 1.4 per block, five- to sixfold over the stack. This mirrors the forward growth of the activations, 1.22 per block, and has the same cause: on the way back, too, each block adds the signal through its body to the signal that bypasses it.

The two norms of a single layer mean little in isolation. They become informative when compared across depth or across training steps, which the exercises do.

What none of these tools can do is make two different machines agree, or decide whether a difference between two runs matters; for that we compare results across seeds. Nor did we carry out the whole procedure with which the section opened, replaying a failed run from a checkpoint (Section 5.6) with the finder attached, though all its parts are now at hand.

The 8-billion-parameter model of the chapter introduction has not become any smaller: its weights still take 32 GB in fp32. But we can now say how such a model is built, initialized, stored, placed on a device, and observed, each time by a walk over the same tree of modules. The chapters that follow take these tools for granted, beginning with the convolutional networks of Chapter 6.

5.8.4 Summary

Every random draw in a training run comes from a generator fixed by a seed. Seeding each generator the run uses fixes the draws, and private generator objects or explicit keys keep unrelated draws from shifting them. Seeding does not fix the arithmetic: floating-point addition depends on the order of the terms, and some GPU operations add in a varying order, so bitwise repetition also needs deterministic operations and holds only on one machine and one software version. It is a tool for debugging; scientific conclusions should hold across seeds. Hooks, or their equivalents in each framework, observe the outputs and gradients of every layer of an unmodified model, which turns the question of where the first NaN appeared into one forward pass.

5.8.5 Exercises

1. code **Block ablation.** A forward hook that returns a value *replaces* the module's output.
    1. Use such a hook to zero the output of one residual block's body, which turns the block into the identity, and measure the relative change in `net`'s output on a fixed batch. Repeat for every block, and rank the blocks by this change.

   2. Rebuild `net` three more times with new random weights and repeat the ranking. State whether the conclusion "block k matters most" survives, and what evidence such a claim would need.

2. code **Residual variance budget.** Let $\mathbf{x}_k$, $\mathbf{b}_k$ and $\mathbf{h}_k$ denote the input, body output and output of block k, with each variance taken over all entries. On one batch, record for every `ResidualBlock` the variance of $\mathbf{x}_k$ with `register_forward_pre_hook` (the pre-hook receives the argument tuple, so unpack it), of $\mathbf{h}_k$ with a forward hook on the block, and of $\mathbf{b}_k$ with a forward hook on its `body`. The standard deviations in the table of Section 5.8.3.1 grow because each block adds its body's output to the stream. Measure how closely

$$\mathrm{Var}[\mathbf{h}_k] = \mathrm{Var}[\mathbf{x}_k] + \mathrm{Var}[\mathbf{b}_k] \tag{5.13}$$

   holds for every block, and name the term this equation omits.

3. code **Seeds and data loading.** ▯ Extend `train_once` to load its data through a `DataLoader` with `shuffle=True` and `num_workers=4`, keeping the single `torch.manual_seed(seed)` call at its top. The dataset adds noise to each example inside `__getitem__`, drawn from an `np.random.default_rng()` object created in the dataset's constructor. Run the code as a standalone script, because worker processes started with `spawn`, the default on macOS and Windows, cannot import a dataset class defined in a notebook.

   1. Predict whether two runs with the same seed agree on the shuffle order and on the noise, and whether the four workers draw the same noise. Justify each prediction from where that randomness is drawn, then verify all three.
   2. Add the `seed_worker` of this section and a seeded `generator` to the loader, and verify that two runs now agree and that the workers draw different noise. Does an example receive the same noise in the second epoch as in the first? Explain the answer from when the loader draws its base seed and calls the initializer.

   Adapted from the PyTorch documentation Multi-process data loading.

4. code **Divergence monitor.** ▯ Turn the finiteness check of Section 5.8.3.2 into a per-step training monitor. Rebuild `net` and train it with SGD at learning rate 1 on the mean squared error between its output and standard normal targets, using one fixed batch of 256 standard normal inputs. Report the first step at which a layer's output is non-finite, and the first such layer by depth. Check the loss at every step as well, and report whether it becomes non-finite before any layer output does.

5. code **Gradient norms.** ▯ Rebuild `net`, multiply the weight of every linear layer inside its residual blocks by 3, and use the mean squared output on a fixed batch as the loss.

   1. Record the norm of each block's output gradient as in this section. After the backward pass, compute each block's parameter-gradient norm, and plot both families against depth.
   2. Express the gradient of a linear layer's weight through the layer's input and its output gradient. Use this expression to explain why the two depth profiles differ, and state which family to monitor for an exploding backward signal.

6. code **Hook coverage.** Attach a forward hook to every leaf module of `torchvision.models.resnet18()` through `named_modules()`, and count how often each hook fires in one forward pass. Explain any module that fires more than once, and name a computation in `BasicBlock.forward` that no module hook observes.

Chapter 6

Convolutional Neural Networks

Every image classifier in this book so far has treated an image as a list of numbers. The MLP of Chapter 4 reads a Fashion-MNIST picture of a shirt or a sneaker as 784 pixel values, and the order of the list does not matter to it: if we shuffled the pixels of every image in the same way and retrained, it would do as well as before. To us the shuffled pictures would be noise. We recognize a sneaker by its outline, and an outline is a pattern among neighboring pixels. It also means the same thing wherever in the image it appears. What would a layer look like that knew these two facts about images?

The visual cortex offers a hint. Hubel and Wiesel found that neurons in the early stages of the visual cortex of cats and monkeys respond to oriented edges and bars, each within a small region of the visual field, the neuron's *receptive field* (Hubel and Wiesel 1959, 1962, 1968). Field (1987) later modeled these responses on natural images with a few small filters, each applied at every position of the image (Figure 6.1). Every filter responds to one local edge pattern, wherever in the image it occurs.

Fukushima's Neocognitron (Fukushima 1980) modeled these cells as a network of local detectors repeated across the image. LeCun and his colleagues trained networks of this kind by backpropagation to read handwritten zip codes (LeCun et al. 1989, 1995). Features computed by deeper layers of networks trained on image classification correspond to responses in later stages of the visual system (Kuzovkin et al. 2018). We should not make too much of this. The analogy motivated the design. It does not show that a particular network works like the brain.

The layer at the center of these networks, the *convolution*, applies one small set of weights to every neighborhood of the image. Each output then depends only on nearby pixels, and a shifted input gives a shifted output, which are the two facts of our opening question. A network that need not relearn a pattern at every position also needs far fewer parameters. *Convolutional neural networks* (CNNs) brought the jump in accuracy on ImageNet (Deng et al. 2009; Krizhevsky et al. 2012) that started the current era of deep learning. Vision transformers (Chapter 11) now share the field with them.

We derive the layer before we use it. Section 6.1 starts from the two facts and finds that they leave little choice: a fully connected layer made to respect both is a convolution. Section 6.2 computes with the layer and lets it learn an edge detector from data. Three sections then supply the controls of a practical layer: padding, stride, and dilation, which set the size of the output and how far each output sees (Section 6.3); channels, several in and several out

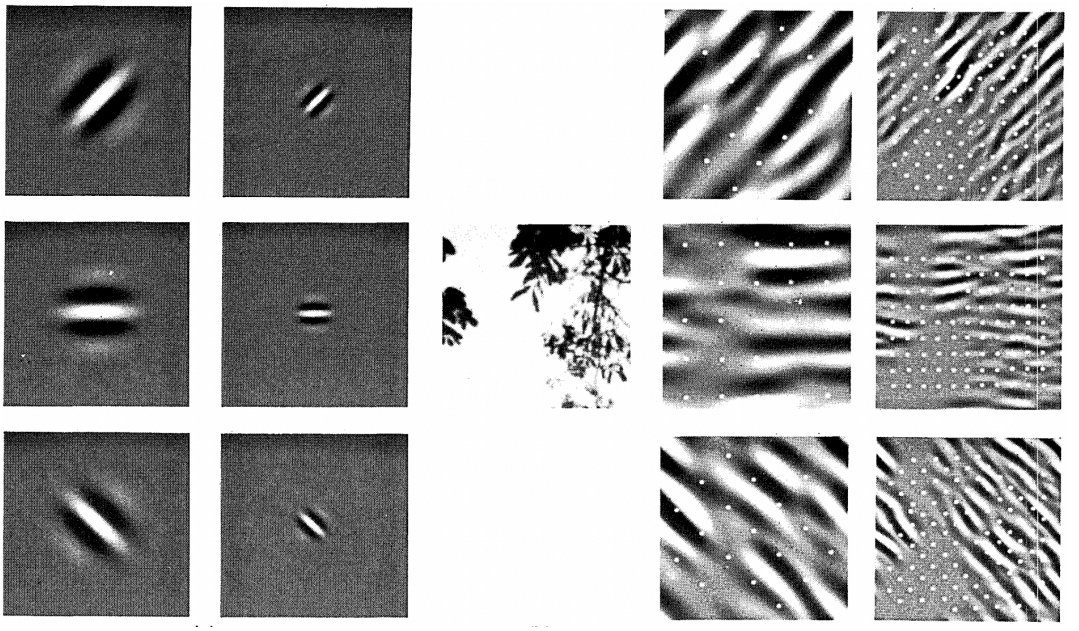


Figure 6.1: Six receptive-field filters and their responses to a natural image, adapted from Field (1987) . The left panel shows the filters, the middle panel the input image, and the right panel the filtered responses sampled at intervals proportional to each filter's size. Different filters respond to different local edge patterns.

(Section 6.4); and pooling, which summarizes a neighborhood so that small shifts do not matter (Section 6.5). Section 6.6 assembles a complete convolutional classifier, compares it with the MLP on the clothing images, and then shuffles the pixels to see how much of its advantage was knowledge of the grid.

The chapter treats images and the classic building blocks; the deeper architectures and training methods that made CNNs dominant follow in Chapter 7.

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.

Books

- Deep Learning — Goodfellow, Bengio & Courville — free HTML; Chapter 9 develops sparse interactions, parameter sharing, and equivariance in detail (Section 6.1, Section 6.5).
- Understanding Deep Learning — Simon J. D. Prince — free PDF; Chapter 10 covers convolutions, stride, dilation, and receptive fields with unusually good figures (Section 6.2, Section 6.3).
- Neural Networks and Deep Learning — Michael Nielsen — free online; Chapter 6 introduces convolutional layers, shared weights, and pooling from first principles (Section 6.2, Section 6.5).

Courses and video lectures

- Stanford CS231n: Deep Learning for Computer Vision — free notes; the "Convolutional Networks" module (layer arithmetic, parameter counting, layer patterns) is the most widely used online companion to this chapter (Section 6.3, Section 6.4).
- Michigan EECS 498-007: Deep Learning for Computer Vision — Justin Johnson — free lecture videos; the convolution and pooling lectures work through kernel/stride/padding arithmetic on the board at exactly this chapter's pace (Section 6.2, Section 6.5).

Tutorials, notes, and interactive

- A guide to convolution arithmetic for deep learning — Dumoulin & Visin (2016) — free, with the famous animations; the definitive visual reference for padding, stride, dilation, and transposed convolutions (Section 6.3).
- CNN Explainer — Wang et al. (Polo Club) — free, zero-install; an interactive convnet running in the browser where you can inspect every activation, kernel, and receptive field (Section 6.4, Section 6.6).
- Computing Receptive Fields of Convolutional Neural Networks — Araujo, Norris & Sim (2019), *Distill* — free; derives the closed-form receptive-field arithmetic that Section 6.2 introduces, including strided and multi-path cases.
- Conv Nets: A Modular Perspective — Chris Olah — free; a short classic building the "convolution as structured weight sharing" intuition (Section 6.1).
- Image Kernels Explained Visually — Victor Powell — free, interactive; slide hand-designed kernels over an image, a perfect warm-up for the edge-detection example in Section 6.2.
- Feature Visualization — Olah, Mordvintsev & Schubert (2017), *Distill* — free; what convnet units actually respond to, a companion to the activation-visualization exercise in Section 6.6.

Foundational papers

- Gradient-Based Learning Applied to Document Recognition — LeCun, Bottou, Bengio & Haffner (1998), *Proc. IEEE* — free PDF; the LeNet-5 paper behind Section 6.6, still worth reading for its systems-level completeness.
- Backpropagation Applied to Handwritten Zip Code Recognition — LeCun et al. (1989), *Neural Computation* — the first convnet trained end-to-end with backprop (paywalled, noted; widely reproduced online) (Section 6.6).
- Neocognitron — Fukushima (1980), *Biological Cybernetics* — the pre-backprop ancestor of alternating convolution/pooling stages (paywalled, noted) (Section 6.5).
- Receptive fields, binocular interaction and functional architecture in the cat's visual cortex — Hubel & Wiesel (1962), *J. Physiology* — free (PMC); the biological root of local receptive fields and hierarchical feature detection (Section 6.1).

6.1 From Fully Connected Layers to Convolutions

As the chapter introduction noted, the MLP of Chapter 4 treats an image as a list of numbers in no particular order. For tabular data, whose columns have no natural order, this is appropriate. For images it discards the most useful thing we know about them: which pixels are neighbors.

The waste shows up in the number of parameters. Consider a classifier that tells cats from

dogs in one-megapixel grayscale photographs. Each input has 10^6 dimensions, so even a hidden layer of only 1000 units needs $10^6 \times 10^3 = 10^9$ weights. That is 4 GB in 32-bit floating point, and far more than any collection of labeled pet photographs could determine. People nevertheless tell cats from dogs with ease, because photographs are far from arbitrary vectors: nearby pixels are related, and a whisker looks like a whisker wherever it appears in the photograph.

Which layer do these two observations leave us with? To find out, we turn them into constraints on a fully connected layer. The hidden units of an MLP have no location, so we first give the hidden layer one unit per pixel, a map of responses the size of the image. For a 1000×1000 image this layer has 10^{12} weights, which looks like a step in the wrong direction. Requiring that a shifted input produce a shifted output reduces this to about 4×10^6 weights. Requiring that each hidden unit look only at a small neighborhood reduces it to a few hundred. The layer that remains is a convolution.

6.1.1 Equivariance and Invariance

The children's game "Where's Waldo" (Figure 6.2) shows what a vision system needs. Each picture contains a crowd of people and activities, and somewhere among them is Waldo. What Waldo looks like does not depend on where he stands. A detector can therefore scan the scene and give every patch a score for how likely it is to contain Waldo, using the same scoring rule for every patch. Many object detection (Girshick et al. 2014) and semantic segmentation (Long et al. 2015) systems work in this way.



Figure 6.2: Can you find Waldo (image courtesy of William Murphy (Infomatique))?

This suggests two requirements for the earliest layers of a vision network:

1. They should respond in the same way to the same patch, wherever it appears, so that a shifted input gives a shifted output.
2. They should look only at small regions and ignore the content of distant ones. This is the *locality* principle.

Deeper layers can then combine these local responses into features of longer range, and eventually into a prediction for the whole image.

To state the first requirement precisely, let T_v shift an image by an offset $v = (v_1, v_2)$, so

that the pixel at (i, j) moves to $(i + v_1, j + v_2)$: $[T_v\mathbf{X}]_{i,j} = [\mathbf{X}]_{i-v_1,j-v_2}$. A function f that maps an image to an image is *translation equivariant* if shifting its input shifts its output by the same amount. A function that maps an image to a prediction is *translation invariant* if shifting its input leaves the output unchanged:

$$\begin{aligned} f(T_v\mathbf{X}) &= T_v f(\mathbf{X}) \quad \text{(equivariance)}, \\ f(T_v\mathbf{X}) &= f(\mathbf{X}) \quad \text{(invariance)}. \end{aligned} \tag{6.1}$$

Figure 6.3 draws the first condition as a square of four images. Its f compares a small pattern with every window of the image, which is the kind of layer this section will arrive at. Whether we shift first and compare afterwards, or compare first and shift the result, we end in the same corner.

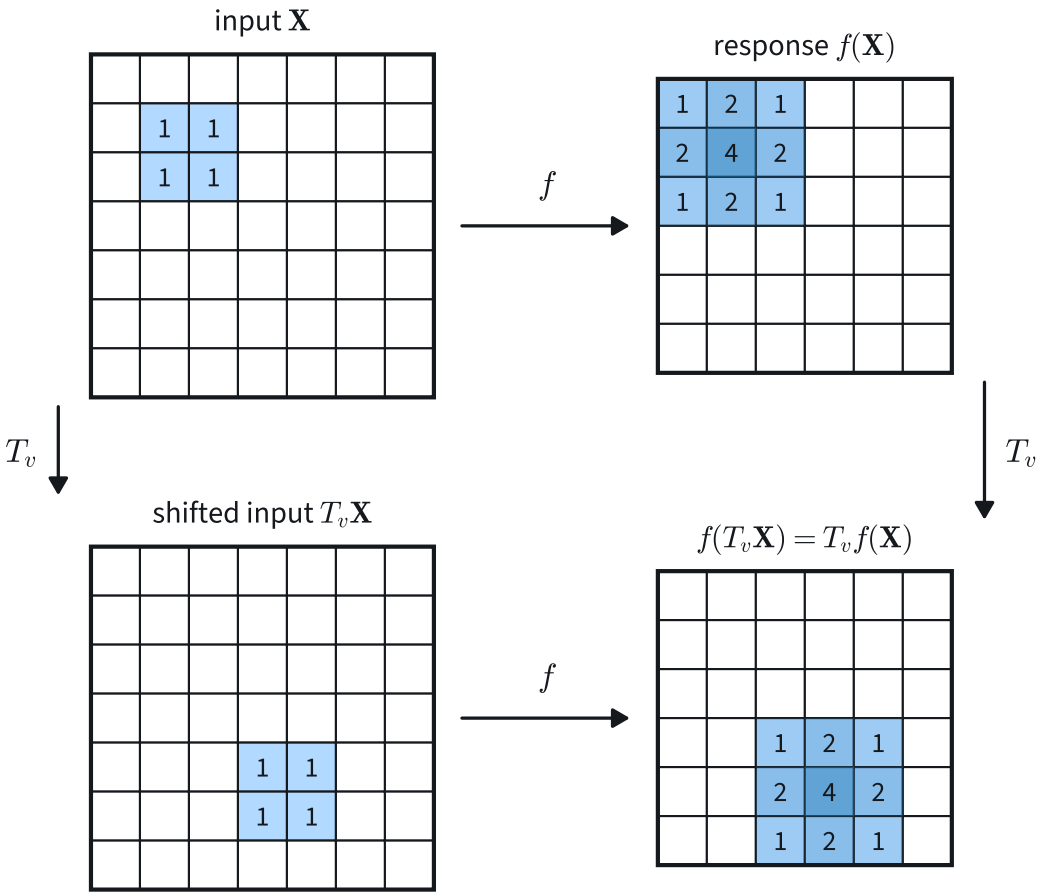


Figure 6.3: Translation equivariance. The input holds a 2×2 patch of ones, and f compares every 2×2 window with that patch, so the response is 4 where a window covers the whole patch. Shifting the input by $v = (3, 2)$ and then applying f (bottom row) gives the same map as applying f and then shifting the response (right column).

Equivariance keeps track of *where* a pattern is, and invariance discards it. A Waldo detector should be equivariant: its map of scores should move with Waldo. The answer to the ques-

tion "is Waldo in the scene?" should be invariant. We obtain it in a final step that aggregates the equivariant map over all locations, for instance by taking its maximum.

6.1.2 Constraints on the Fully Connected Layer

Both constraints are easiest to see in one dimension, where a weight matrix is something we can draw. Take a signal of eight samples $x_0, \dots, x_7$ and a fully connected layer with one hidden unit per sample, $h_i = u_i + \sum_k W_{i,k} x_k$. Its weights form an 8×8 matrix with 64 free entries, panel (a) of Figure 6.4.

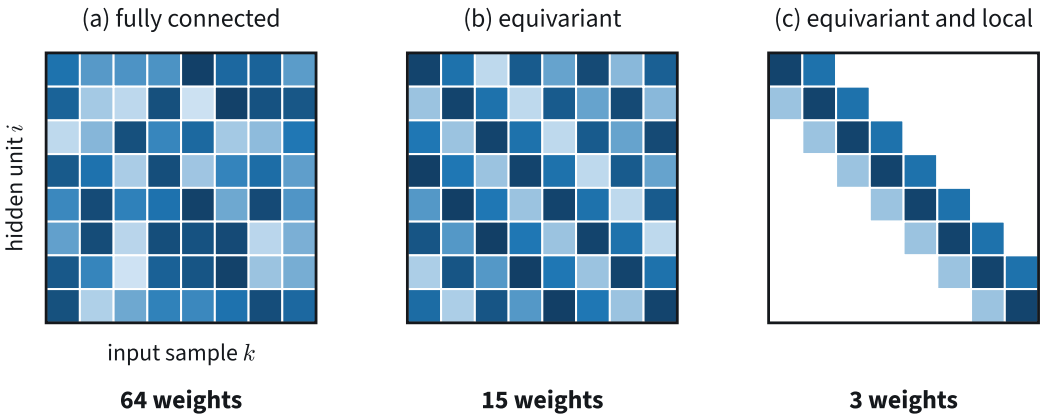


Figure 6.4: Weights of a linear layer that maps a one-dimensional signal of 8 samples (columns) to 8 hidden units (rows). Equal colors are equal weights; white is zero. (a) Fully connected: every weight is free. (b) Translation equivariant: a weight depends only on the offset between input and output, so the matrix is constant along each diagonal. (c) Equivariant and local: only the offsets -1, 0, and 1 have nonzero weights.

What does equivariance demand of this matrix? Set the biases aside (the proof below shows that they must all be equal) and feed the layer a signal with a single 1 at sample k. The response is column k of $\mathbf{W}$. Move the 1 to sample $k+1$, and the response is column $k+1$. Equivariance requires the second response to be the first one moved down by one unit. Every column is therefore its left neighbor moved down by one, $W_{i+1,k+1} = W_{i,k}$, and the matrix is constant along each diagonal, as in panel (b). A diagonal holds the weights with one offset $a = k - i$ between sample and hidden unit, so a weight depends only on this offset. Eight samples allow offsets from -7 to 7, and 15 weights remain of 64.

Locality removes most of the 15. If a hidden unit may depend only on samples within distance 1, every offset other than -1, 0, and 1 gets weight zero. Panel (c) has 3 weights, the same three in every row: the layer compares one small pattern with the signal at every position. This is the whole derivation. The rest of the section repeats it with more indices.

For images nothing changes except that positions and offsets become pairs. Let the input image $\mathbf{X}$ and the hidden representation $\mathbf{H}$ be matrices of the same shape, with pixel $[\mathbf{X}]_{k,l}$ at location (k, l) and hidden unit $[\mathbf{H}]_{i,j}$ at location (i, j). A weight now needs four indices, two for the hidden unit and two for the pixel, so the weights form a fourth-order tensor W. With a matrix of biases $\mathbf{U}$, the layer computes

$$[\mathbf{H}]_{i,j} = [\mathbf{U}]_{i,j} + \sum_k \sum_l [\mathsf{W}]_{i,j,k,l}[\mathbf{X}]_{k,l}.$$

Both constraints concern where a pixel lies *relative to* the hidden unit. It is therefore convenient to name each weight by its offsets $a = k - i$ and $b = l - j$ from the hidden unit, and to write $[\mathsf{V}]_{i,j,a,b} = [\mathsf{W}]_{i,j,i+a,j+b}$:

$$[\mathbf{H}]_{i,j} = [\mathbf{U}]_{i,j} + \sum_a \sum_b [\mathsf{V}]_{i,j,a,b}[\mathbf{X}]_{i+a,j+b}. \tag{6.2}$$

The offsets take positive and negative values. Nothing has been computed here: the layer is the same, and we have only read its weights along the diagonals, as in the one-dimensional case. With one hidden unit for each of the 10^6 pixels of a 1000×1000 image, this single layer has $10^6 \times 10^6 = 10^{12}$ weights.

6.1.2.1 Translation Equivariance

The argument with a single 1 carries over to images: a layer of the form Equation 6.2 is equivariant exactly when its weights and its bias do not depend on the location (i, j) of the hidden unit. That shared weights give equivariance is a one-line check. The converse matters more: equivariance leaves no other choice.

To state this precisely we must say what a shift does at the border of an image, where content leaves the frame and nothing enters it. We avoid the question by placing the image on an infinite grid of pixels, with zeros everywhere outside the image, so that a shift loses nothing. On a finite image equivariance holds only away from the border, a point to which Section 6.3 returns.

Proposition. Let the input, with finitely many nonzero pixels, and the hidden representation live on the infinite grid. The layer Equation 6.2 is translation equivariant for every shift v if and only if there are a scalar u and a matrix $\mathbf{K}$ such that $[\mathbf{U}]_{i,j} = u$ and $[\mathsf{V}]_{i,j,a,b} = [\mathbf{K}]_{a,b}$ for all i, j, a, b.

Proof. If neither the weights nor the bias depend on (i, j), shifting the input by v shifts the output by v:

$$[f(T_v\mathbf{X})]_{i,j} = u + \sum_a \sum_b [\mathbf{K}]_{a,b}[\mathbf{X}]_{i-v_1+a,j-v_2+b} = [f(\mathbf{X})]_{i-v_1,j-v_2} = [T_v f(\mathbf{X})]_{i,j}.$$

Conversely, suppose that the layer is equivariant. The all-zero image is unchanged by every shift, and the layer maps it to $\mathbf{U}$. Hence $\mathbf{U} = T_v\mathbf{U}$ for every v, and the bias is a constant u. Subtracting u from both sides of the equivariance condition shows that the weighted sum alone is equivariant too. Now feed it the image with a single 1 at pixel (k, l) and zeros elsewhere. Its response at (i, j) is the weight $[\mathsf{W}]_{i,j,k,l}$. Shifting this image by v moves the 1 to $(k + v_1, l + v_2)$, and equivariance requires the response to shift by v as well: $[\mathsf{W}]_{i,j,k+v_1,l+v_2} = [\mathsf{W}]_{i-v_1,j-v_2,k,l}$. Choosing $(k, l) = (a, b)$ and $v = (i, j)$ gives $[\mathsf{V}]_{i,j,a,b} = [\mathsf{W}]_{i,j,i+a,j+b} = [\mathsf{W}]_{0,0,a,b}$, which does not depend on (i, j). ■

In words, a single bright pixel reveals the weights, as the single 1 did in one dimension, and a tensor V that does not depend on (i, j) is the matrix that is constant along its diagonals. With shared weights the layer simplifies to

$$[\mathbf{H}]_{i,j} = u + \sum_a \sum_b [\mathbf{K}]_{a,b} [\mathbf{X}]_{i+a,j+b}. \tag{6.3}$$

Every hidden unit computes the same weighted sum of the pixels around it, so the layer compares one pattern, $\mathbf{K}$, with the image at every location. For a 1000×1000 image surrounded by zeros, only the offsets a and b from -999 to 999 ever meet a nonzero pixel, so $\mathbf{K}$ has $1999^2 \approx 4 \times 10^6$ weights instead of 10^{12}.

6.1.2.2 Locality

Four million weights are still too many for one pattern, and most of them tie a hidden unit to distant pixels. We therefore impose locality: the hidden unit at (i, j) should depend only on pixels within distance Δ of (i, j) along each axis. This means $[\mathbf{K}]_{a,b} = 0$ whenever $|a| > \Delta$ or $|b| > \Delta$, and the layer becomes

$$[\mathbf{H}]_{i,j} = u + \sum_{a=-\Delta}^{\Delta} \sum_{b=-\Delta}^{\Delta} [\mathbf{K}]_{a,b} [\mathbf{X}]_{i+a,j+b}. \tag{6.4}$$

Each hidden unit is a weighted sum of the $(2\Delta+1) \times (2\Delta+1)$ window of pixels centered at its own location, with the same weights at every location. The layer has $(2\Delta + 1)^2$ weights and one bias: 9 weights for $\Delta = 1$ and 121 for $\Delta = 5$. Typical values of Δ are below 10, so at most a few hundred weights replace the 4×10^6 of the shared layer, while the hidden representation keeps the size of the image.

Equation Equation 6.4 defines a *convolutional layer*. Networks that contain convolutional layers are called *convolutional neural networks* (CNNs). The matrix $\mathbf{K}$ is the layer's *convolution kernel*, also called its *filter*, and its entries are the layer's weights. Among the first models to share weights across positions in this way were time-delay neural networks for speech recognition (Waibel et al. 1989) and the shift-invariant networks of Zhang et al. (1988) .

The reduction from 10^{12} to a few hundred parameters comes from restricting the class of functions, not from a more economical way to represent the same functions. A convolutional layer can represent a mapping only if that mapping is local and translation equivariant. Such assumptions, made before any data are seen, are the layer's inductive bias (Section 4.5). When they match the data, as they do for edges and textures in natural images, the model needs far fewer examples and generalizes better. When they do not, the model may fail even to fit its training data. In a passport photograph, a dark blob in the upper half is an eye and a similar one lower down is a nostril, yet a convolutional layer must respond to both alike. A single layer also sees only a window of width $2\Delta + 1$. Stacking convolutional layers with nonlinearities between them lets deeper units depend on larger regions of the image and represent larger structures.

6.1.3 Convolution and Cross-Correlation

The name comes from mathematics, and it does not quite fit. The *convolution* of two functions on the integers is $(f * g)(i) = \sum_a f(a) g(i - a)$ (for functions on $\mathbb{R}^d$ an integral takes

the place of the sum). It measures the overlap between f and a copy of g that is flipped and shifted by i. For functions on a two-dimensional grid it becomes

$$(f * g)(i, j) = \sum_a \sum_b f(a, b) g(i - a, j - b). \tag{6.5}$$

The layer Equation 6.4 reads the image at $(i+a, j+b)$ rather than at $(i-a, j-b)$. Substituting $-a$ for a and $-b$ for b in its sum shows that it computes the convolution of the image with the flipped kernel $[\mathbf{K}']_{a,b} = [\mathbf{K}]_{-a,-b}$:

$$\sum_a \sum_b [\mathbf{K}]_{a,b} [\mathbf{X}]_{i+a,j+b} = \sum_a \sum_b [\mathbf{K}']_{a,b} [\mathbf{X}]_{i-a,j-b} = (\mathbf{K}' * \mathbf{X})(i, j).$$

The operation without the flip is called *cross-correlation*. Does the difference matter? For a learned kernel it does not: a layer that cross-correlates with $\mathbf{K}$ and a layer that convolves with $\mathbf{K}'$ represent the same functions. Deep learning libraries compute cross-correlation and call it convolution, and this book follows that usage.

6.1.4 Channels

So far the input and the hidden representation were single matrices. A color image has three values per pixel, its red, green, and blue intensities. It is therefore a third-order tensor X of shape 3 × height × width, with entries $[\mathsf{X}]_{c,i,j}$. The first axis is the *channel* axis, and the other two are spatial, so every location carries a vector of values. A kernel for such an input has one weight for every input channel and every offset, and the hidden unit at (i, j) sums over all input channels and over the window.

One kernel produces one map of responses and detects one kind of pattern. A kernel that matches Waldo's appearance yields a map that peaks where he stands (Figure 6.5). A useful layer detects many patterns at once: edges in different directions, textures, colors. The hidden representation therefore becomes a third-order tensor H as well, a stack of maps with one *output channel* per pattern. Each output channel o has its own kernel and its own bias u_o, which gives the kernel a fourth index:

$$[\mathsf{H}]_{o,i,j} = u_o + \sum_c \sum_{a=-\Delta}^{\Delta} \sum_{b=-\Delta}^{\Delta} [\mathsf{K}]_{o,c,a,b} [\mathsf{X}]_{c,i+a,j+b}. \tag{6.6}$$

Here c runs over the input channels and o names the output channel. We write the channel indices first, the output channel before the input channel, and keep this order for the rest of the chapter. Each output channel is a spatial map of one learned feature, which is why the channels of a hidden representation are also called *feature maps*. The next layer takes the whole stack as its input. With c_i input channels and c_o output channels the layer has $(2\Delta + 1)^2 c_\mathrm{i} c_\mathrm{o}$ weights and c_o biases. A layer with a 3×3 kernel from an RGB image to 64 channels has $9 \cdot 3 \cdot 64 + 64 = 1{,}792$ parameters, whatever the size of the image.

Nothing in this derivation depends on the grid having two axes. For the samples of an audio signal or the tokens of a text the kernel slides along one axis, and for the frames of a video or a medical scan along three.



Figure 6.5: A Waldo detector: the response of a kernel that matches Waldo peaks at his location (image courtesy of William Murphy (Infomatique)).

Equation Equation 6.6 is the convolutional layer on which the rest of the chapter builds. The derivation was carried out on an infinite grid with a single window, and it has left three matters unsettled. A finite image has borders, where the window does not fit. Deeper units should see more of the image than one window. And the network must eventually turn its maps into a single answer, such as whether Waldo is *anywhere* in the scene. Before we take these up, we should compute Equation 6.4 once by hand and in code.

6.1.5 Summary

Treating an image as an unordered vector makes a fully connected layer enormous: mapping a megapixel image to a hidden representation of the same size takes 10^{12} weights. Two assumptions about images reduce this to a few hundred. Translation equivariance forces every location to use the same kernel, as we proved for layers of this form on an infinite grid, and locality restricts the kernel to a small window. The resulting layer computes a cross-correlation, which deep learning calls a convolution, and channels let one layer detect many patterns at once. The savings come from restricting the function class: a convolutional layer cannot give the same local pattern different responses at different locations.

6.1.6 Exercises

1. **Weight sharing on a ring.** Repeat the argument of Figure 6.4 for a signal of four samples on a periodic grid, where the sample after the last one is the first one again, so that a shift loses nothing.
    1. Feed the layer a single 1 at each sample in turn and write down the 4×4 weight matrix of a translation-equivariant layer. How many free weights does it have? Compare with the $2 \cdot 4 - 1 = 7$ offsets of four samples surrounded by zeros.
    2. Impose locality with offsets -1, 0, and 1. Which entries of the matrix are nonzero, and which of them have no counterpart in panel (c) of the figure?
2. **Commutativity of convolution.**
    1. Prove that the convolution of Equation 6.5 is commutative, i.e., $f * g = g * f$.
    2. Define the cross-correlation $(f \star g)(i, j) = \sum_a \sum_b f(i + a, j + b)\, g(a, b)$, the

operation of Equation 6.4 with input f and kernel g. Express $(g \star f)(i, j)$ through values of $f \star g$, and decide from this expression whether cross-correlation is commutative.

3. **Convolutions for audio.** Audio data is often represented as a one-dimensional sequence of samples.
   1. Derive the one-dimensional analog of Equation 6.4.
   2. Name one audio task for which a layer that is local and translation equivariant in time is a good fit, and one for which it is not. Justify each choice.
   3. A spectrogram represents a recording as a two-dimensional array whose axes are time and frequency, with frequency on a linear scale in hertz. For each axis, state whether translation equivariance is a reasonable assumption, and justify your answer from what a shift along that axis does to the sound. Then answer the same question for a frequency axis on a logarithmic scale.
4. **Convolutions on text.** Represent a sentence as a sequence of word vectors, so that each word occupies one position and the entries of its vector act as channels. Apply one-dimensional convolutional layers, the analog of Equation 6.6, with nonlinearities in between.
   1. Take $\Delta = 1$. In the sentence "The movie that my friends recommended last week was not good," the verb "was" agrees with "movie." How many stacked layers are needed before a single output depends on both words? How does this number grow with the distance between two words?
   2. Apply a single layer followed by a ReLU and sum its output over all positions to obtain a sentence representation. State whether "the dog bit the man" and "the man bit the dog" receive the same representation for $\Delta = 0$ and for $\Delta = 1$. Which word-order information does this representation keep, and which does it discard?
5. **Equivariance versus invariance.** A convolutional classifier computes feature maps $h(\mathsf{X})$ with stacked layers of Equation 6.6 and nonlinearities in between, and then a prediction $g(h(\mathsf{X}))$ with a head g.
   1. Assume a periodic grid, which wraps around so that the column after the last one is the first one again; on it, h is translation equivariant by the argument of the proposition. Let g average each channel of $h(\mathsf{X})$ over all locations. Using Equation 6.1, show that the prediction is translation invariant.
   2. An image shows a single digit in one of the nine cells of a 3×3 grid, and the task is to name the cell. Explain why the classifier of the first part cannot solve this task. Describe a head g that solves it from the same feature maps $h(\mathsf{X})$.
6. **Boundary effects.** A $\Delta = 1$ kernel, i.e., a 3×3 kernel, is applied to an $n \times n$ image only at the positions where the whole window lies inside the image, so the output has $(n-2)^2$ positions.
   1. Count the output positions to which each of three input pixels contributes: the corner pixel $(0, 0)$, a boundary pixel $(0, j)$ with $2 \le j \le n-3$, and an interior pixel (i, j) with $2 \le i, j \le n-3$, where indices run from 0 to $n-1$. Which fraction of the output positions uses at least one pixel of the outermost ring, for $n = 28$ and for $n = 5$?
   2. Stack L such layers on an image with odd $n \ge 4L + 1$. To how many outputs of the last layer does a corner pixel of the input contribute, and to how many does the center pixel contribute?
7. **Rotation equivariance.** ⊠ The proposition of this section derived weight sharing from equivariance to shifts. Ask the same of rotations. On the infinite grid of the proposition, let R rotate an image by 90° about the origin, $[R\mathbf{X}]_{i,j} = [\mathbf{X}]_{-j,i}$.

1. Show that the convolutional layer Equation 6.4 satisfies $f(R\mathbf{X}) = Rf(\mathbf{X})$ for every image $\mathbf{X}$ if and only if $[\mathbf{K}]_{a,b} = [\mathbf{K}]_{-b,a}$ for all offsets (a, b).
2. For $\Delta = 1$, how many free weights does such a kernel have? Show that its response to an image does not change when the image is rotated by 90° about the pixel being evaluated, so that it cannot tell a vertical edge from a horizontal one.
3. Instead of constraining one kernel, use four output channels whose kernels are copies of one unconstrained kernel $\mathbf{K}$, rotated by 0°, 90°, 180°, and 270°. Show that rotating the input rotates every output map and permutes the four channels cyclically. What does this layer keep that the constrained kernel of the first part loses?

6.2 Convolutions for Images

Section 6.1 derived the convolutional layer on paper, from two requirements on a fully connected layer: that a shifted input produce a shifted output, and that each output depend only on a small neighborhood. The derivation ended with a formula. What does that formula compute on an actual image, and can a network find good weights for it by itself?

Finding the vertical edges in a grayscale image, the places where a bright region meets a dark one, is about the simplest instance of both questions. Picture a small white image with a black vertical band down its middle. If we subtract from each pixel its right-hand neighbor, the difference is zero inside flat regions and nonzero where the brightness changes, positive where white turns to black and negative where black turns to white. An edge can occur in any row and any column, so the same detector, the two weights $(1, -1)$, must be applied at every position. Sliding it over the image gives one output for every position at which it fits, and these outputs form a map of the edges. This is the operation that a convolutional layer computes: a small array of weights, the kernel, is compared with every window of the image, and each comparison yields one output.

Designing a detector by hand works for vertical edges. A network, however, needs thousands of detectors, and nobody knows in advance which patterns they should match. We therefore define the operation, implement it, apply it to the edge example, and then let gradient descent find the kernel $(1, -1)$ from the image and its edge map alone. Two shorter topics follow: how the operation is computed efficiently, and how much of the image a unit in a stack of such layers sees.

6.2.1 Cross-Correlation

6.2.1.1 The Operation

The edge detector uses a single kernel of shape 1×2. Let us first state the operation for any input and any kernel, omitting channels for now: the input $\mathbf{X}$ is a matrix of height n_h and width n_w, and the kernel $\mathbf{K}$ is a matrix of height k_h and width k_w. The kernel's shape is also the shape of the *window*, the part of the input that the kernel covers at one position. Place the window with its upper-left corner at input position (i, j), multiply the input elements under it elementwise with the kernel, and sum. The result is the output at position (i, j):

$$[\mathbf{Y}]_{i,j} = \sum_{a=0}^{k_\mathrm{h}-1} \sum_{b=0}^{k_\mathrm{w}-1} [\mathbf{K}]_{a,b} \, [\mathbf{X}]_{i+a,j+b}. \tag{6.7}$$

This is the *two-dimensional cross-correlation* of $\mathbf{X}$ with $\mathbf{K}$, which we write $\mathbf{Y} = \mathbf{X} \star \mathbf{K}$. Indices start at 0 here, as in code. The layer of Section 6.1 indexed the window by offsets from $-\Delta$ to Δ around its center instead. The two conventions differ only by a shift of the output by Δ positions, but the corner is the more convenient anchor: it also serves kernels that have no center, such as the 2×2 kernel of the next figure and the 1×2 edge detector.

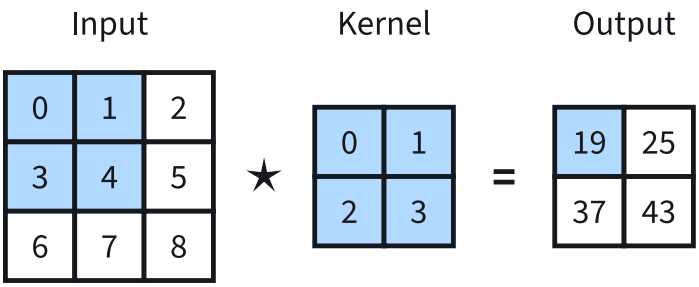


Figure 6.6: Two-dimensional cross-correlation. The shaded portions are the first output element and the input and kernel elements used to compute it: $0\times0+1\times1+3\times2+4\times3 = 19$.

In Figure 6.6 the input has shape 3×3 and the kernel has shape 2×2. The window starts at the upper-left corner of the input and slides one element at a time, from left to right and from top to bottom. The window fits at four positions, which give the four outputs

$$\begin{aligned}
0 \times 0 + 1 \times 1 + 3 \times 2 + 4 \times 3 &= 19,\\
1 \times 0 + 2 \times 1 + 4 \times 2 + 5 \times 3 &= 25,\\
3 \times 0 + 4 \times 1 + 6 \times 2 + 7 \times 3 &= 37,\\
4 \times 0 + 5 \times 1 + 7 \times 2 + 8 \times 3 &= 43.
\end{aligned}$$

Note that the output is smaller than the input, since the window must lie inside the input. Along each axis it fits at $n - k + 1$ positions, so the output has shape

$$(n_\mathrm{h} - k_\mathrm{h} + 1) \times (n_\mathrm{w} - k_\mathrm{w} + 1). \tag{6.8}$$

The 3×3 input with a 2×2 kernel gives 2×2, the 6×8 edge image with a 1×2 kernel gives 6×7, and a 28×28 Fashion-MNIST image with a 5×5 kernel gives 24×24. If the shrinking is unwelcome, a border of zeros around the input, called *padding*, undoes it; Section 6.3 treats it together with other ways to control the output shape.

6.2.1.2 Implementation

The function `corr2d` computes Equation 6.7 in the plainest way possible, with two loops over the output positions.

```
def corr2d(X, K):
    """Compute 2D cross-correlation."""
    h, w = K.shape
    Y = d2l.zeros((X.shape[0] - h + 1, X.shape[1] - w + 1))
```

```
    for i in range(Y.shape[0]):
        for j in range(Y.shape[1]):
            Y[i, j] = d2l.reduce_sum((X[i: i + h, j: j + w] * K))
    return Y
```

Does `corr2d` reproduce the four outputs of Figure 6.6? We construct its input and kernel and compare.

```
X = d2l.tensor([[0.0, 1.0, 2.0], [3.0, 4.0, 5.0], [6.0, 7.0, 8.0]])
K = d2l.tensor([[0.0, 1.0], [2.0, 3.0]])
corr2d(X, K)
```

```
tensor([[19., 25.],
        [37., 43.]])
```

The output matches the figure, number for number.

6.2.1.3 Convolution and Cross-Correlation

Strictly speaking, `corr2d` does not compute a convolution. The mathematical *convolution* flips the kernel horizontally and vertically before sliding it (Section 6.1). The numbers show how little is at stake. For the input of Figure 6.6, a layer that computes true convolutions produces the same outputs 19, 25, 37, and 43 with the kernel $\mathbf{K}' = \begin{bmatrix} 3 & 2 \\ 1 & 0 \end{bmatrix}$, and since kernels are learned from data, it would simply learn $\mathbf{K}'$ in place of $\mathbf{K}$. From now on we say convolution for both, as the libraries do.

6.2.2 Convolutional Layers

6.2.2.1 The Layer

A convolutional layer cross-correlates its input with a kernel and adds a scalar bias. Its two parameters are the kernel and the bias. The kernel starts from random values, drawn from $\mathcal{N}(0, 0.01^2)$ like the weights of the MLP in Chapter 4, and the bias starts at zero. The class `Conv2D` declares the two parameters in its constructor, and its forward method calls `corr2d` and adds the bias. A layer whose kernel has height h and width w is called an $h \times w$ convolutional layer.

```
class Conv2D(nn.Module):
    def __init__(self, kernel_size):
        super().__init__()
        self.weight = nn.Parameter(0.01 * torch.randn(kernel_size))
        self.bias = nn.Parameter(torch.zeros(1))

    def forward(self, x):
        return corr2d(x, self.weight) + self.bias
```

Unlike a fully connected layer, this layer is not tied to one input size, since its parameters belong to the kernel alone. One 2×2 layer with its five parameters applies to the 3×3 input `X` and to a 6×8 image alike, and returns outputs of shape 2×2 and 5×7.

```
layer = Conv2D((2, 2))
layer(X).shape, layer(d2l.ones((6, 8))).shape
```

```
(torch.Size([2, 2]), torch.Size([5, 7]))
```

6.2.2.2 Edge Detection

It is time to run the edge detector of the opening. The image has 6×8 pixels; its middle four columns are black (0) and the rest is white (1).

```
X = d2l.ones((6, 8))
X[:, 2:6] = 0
X
```

```
tensor([[1., 1., 0., 0., 0., 0., 1., 1.],
        [1., 1., 0., 0., 0., 0., 1., 1.],
        [1., 1., 0., 0., 0., 0., 1., 1.],
        [1., 1., 0., 0., 0., 0., 1., 1.],
        [1., 1., 0., 0., 0., 0., 1., 1.],
        [1., 1., 0., 0., 0., 0., 1., 1.]])
```

The kernel has height 1 and width 2:

$$\mathbf{K} = \begin{bmatrix} 1 & -1 \end{bmatrix}. \tag{6.9}$$

At position (i, j) it computes $[\mathbf{X}]_{i,j} - [\mathbf{X}]_{i,j+1}$, the difference between two horizontally adjacent pixels, which is 0 wherever they are equal. This is a *finite difference*: up to sign, it approximates the derivative of the brightness along the row. A derivative vanishes where nothing changes, which is why the kernel responds only where the brightness does.

```
K = d2l.tensor([[1.0, -1.0]])
```

Cross-correlating `X` with `K` gives the edge map `Y`.

```
Y = corr2d(X, K)
Y
```

```
tensor([[ 0.,  1.,  0.,  0.,  0., -1.,  0.],
        [ 0.,  1.,  0.,  0.,  0., -1.,  0.],
        [ 0.,  1.,  0.,  0.,  0., -1.,  0.],
        [ 0.,  1.,  0.,  0.,  0., -1.,  0.],
        [ 0.,  1.,  0.,  0.,  0., -1.,  0.],
        [ 0.,  1.,  0.,  0.,  0., -1.,  0.]])
```

The output is 1 in the column where white turns to black, -1 in the column where black turns to white, and 0 everywhere else. The same kernel applied to the transposed image, whose band is horizontal, should find nothing.

```
corr2d(d2l.transpose(X), K)
```

```
tensor([[0., 0., 0., 0., 0.],
        [0., 0., 0., 0., 0.],
        [0., 0., 0., 0., 0.],
        [0., 0., 0., 0., 0.],
        [0., 0., 0., 0., 0.],
        [0., 0., 0., 0., 0.],
        [0., 0., 0., 0., 0.],
        [0., 0., 0., 0., 0.]])
```

The output is zero everywhere: $(1, -1)$ detects only vertical edges. Edges in other directions need other kernels, and so do corners, textures, and every other pattern a network may need.

6.2.2.3 Learning a Kernel

We cannot design all of these by hand, least of all for the intermediate layers of a stack, where we do not even know which patterns to ask for. Can gradient descent find a kernel from the input `X` and the desired output `Y` alone? Let us try it on the one kernel whose answer we know. We start from a small random 1×2 kernel, drawn as before, measure the squared error between the layer's output and `Y`, summed over all 42 output positions, and take 10 gradient steps with learning rate 0.03. Instead of `Conv2D` we use the built-in convolutional layer, which computes the same cross-correlation fast. It expects two more axes, one for the examples of a minibatch and one for channels (Section 6.4), so we reshape `X` and `Y` accordingly. We omit the bias, which the edge map does not need.

```
conv2d = nn.Conv2d(1, 1, kernel_size=(1, 2), bias=False)
nn.init.normal_(conv2d.weight, std=0.01)
# Layers take (example, channel, height, width): one example, one channel
X = X.reshape((1, 1, 6, 8))
Y = Y.reshape((1, 1, 6, 7))
lr = 3e-2  # Learning rate

for i in range(10):
    l = ((conv2d(X) - Y) ** 2).sum()
    conv2d.zero_grad()
    l.backward()
    with torch.no_grad():
        conv2d.weight -= lr * conv2d.weight.grad
    if i % 3 == 0:
        print(f'step {i}, loss {l:.3f}')
```

```
step 0, loss 12.216
step 3, loss 0.840
step 6, loss 0.058
step 9, loss 0.004
```

Before the first step the loss is close to 12: the small random kernel outputs nearly 0 everywhere, so each of the 12 edge positions misses its target ± 1 by about 1. After nine steps the loss is below 0.01. The learned kernel is close to $(1, -1)$:

```
d2l.reshape(conv2d.weight.data, (1, 2))
```

```
tensor([[ 0.9891, -0.9876]])
```

Both weights are within about 0.02 of the hand-designed detector. Nothing in the training procedure was specific to edges: given the outputs it should produce, gradient descent finds the kernel. Admittedly, we made its task easy by supplying the edge map. In a deep network nobody supplies the intermediate outputs, and the kernels of every layer are learned from the loss at the end of the network.

6.2.3 Convolution as Matrix Multiplication

The two loops of `corr2d` are easy to read and far too slow for real images and networks, since they visit one output position at a time. A standard remedy is to turn the whole computation into one large matrix product, the operation that GPUs execute best. Every output element in Figure 6.6 is a dot product: the kernel, flattened into a vector of length $k_\mathrm{h}k_\mathrm{w}$, times the input patch under the window, flattened the same way. If we extract each of the $(n_\mathrm{h} - k_\mathrm{h} + 1)(n_\mathrm{w} - k_\mathrm{w} + 1)$ patches that the window visits, flatten it into a row, and stack the rows, we obtain a matrix with one row per output position. The whole cross-correlation then becomes a single product of this patch matrix with the flattened kernel. The rearrangement is called *im2col*, since it turns image patches into the columns (here rows) of a matrix.

The patch matrix takes memory, since each input element is copied into up to $k_\mathrm{h}k_\mathrm{w}$ rows. We build it for the input and kernel of Figure 6.6 and check that one matrix product reproduces `corr2d`.

```
X = d2l.tensor([[0.0, 1.0, 2.0], [3.0, 4.0, 5.0], [6.0, 7.0, 8.0]])
K = d2l.tensor([[0.0, 1.0], [2.0, 3.0]])
h, w = K.shape
h_o, w_o = X.shape[0] - h + 1, X.shape[1] - w + 1
patches = d2l.stack([d2l.reshape(X[i:i + h, j:j + w], (-1,))
                     for i in range(h_o) for j in range(w_o)])
Y_mat = d2l.reshape(d2l.matmul(patches, d2l.reshape(K, (-1, 1))), (h_o, w_o))
Y_mat, d2l.reduce_sum(d2l.abs(Y_mat - corr2d(X, K)))
```

```
(tensor([[19., 25.],
         [37., 43.]]),
 tensor(0.))
```

The sum of absolute differences is zero: the single matrix product and the loops of `corr2d` compute the same output.

We could equally well unroll the kernel instead of the input. The result is a sparse, banded matrix $\mathbf{W}$ that multiplies the flattened image, and we have met it before: it is the two-dimensional counterpart of panel (c) of Figure 6.4, in which every row holds the same few weights. A convolution is a fully connected layer whose weight matrix has this pattern, which is where the derivation of Section 6.1 started. The exercises construct $\mathbf{W}$ explicitly.

6.2.4 Feature Maps and Receptive Fields

A single layer sees only a window the size of its kernel. Recognizing an object requires seeing much more of the image, so we need a way to measure how much of the input a unit sees. The output of a convolutional layer is a feature map (Section 6.1.4); the edge map `Y` is one, for vertical edges. For an element x of a feature map, its *receptive field* is the set of input

pixels that can affect x during forward propagation. Relative to an earlier layer, it is the set of that layer's elements that can affect x. The term comes from neurophysiology, where it names the part of the visual field to which a neuron responds (Figure 6.1).

6.2.4.1 Stacked Layers

In Figure 6.6 the receptive field of the shaded output 19 is the four shaded input elements. Call the 2×2 output $\mathbf{Y}$, and add a second 2×2 convolutional layer that takes $\mathbf{Y}$ as input and outputs a single element z. The receptive field of z on $\mathbf{Y}$ is all four elements of $\mathbf{Y}$, and on the input it is all nine input elements. Stacking layers lets an element respond to a larger region of the input.

How fast does the region grow? A layer with a $k \times k$ kernel reaches $k - 1$ positions beyond a single position of its input, so each layer widens the receptive field by $k - 1$. Starting from one pixel, an element at the top of a stack of L layers with kernel sizes $k_1, \dots, k_L$ has a receptive field of side length

$$r = 1 + \sum_{i=1}^{L} (k_i - 1).$$

For L stacked 3×3 layers each layer adds 2, so the stack sees $(2L + 1) \times (2L + 1)$ input pixels: two layers cover 5×5 and three cover 7×7 (Figure 6.7). This slow, linear growth is no reason to prefer large kernels. Three 3×3 layers match the receptive field of one 7×7 layer with fewer parameters, 27 weights instead of 49 per pair of input and output channels, and with three nonlinearities instead of one.

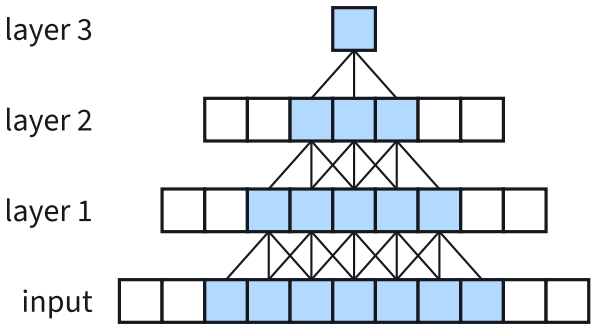


Figure 6.7: Receptive field of one unit after three one-dimensional layers with kernel size 3. Shaded cells are the units that the top unit depends on, and the lines connect each of them to its inputs. Each layer widens the receptive field by 2, to 7 inputs.

6.2.4.2 Effective Receptive Fields

The count r is generous. It gives the theoretical receptive field, the set of inputs that can affect an activation at all, and says nothing about how much each of them does. Luo et al. (2016) measured the *effective* receptive field through gradients: in both randomly initialized and trained networks it concentrates near the center of the theoretical region, with a roughly Gaussian profile. A unit at the top of a deep stack can see far, but it looks mostly at the middle.

Two defects of the layer are left to repair. Every layer shrinks the image and uses the pixels near its border less often than those in the middle, and a single-channel kernel detects only one pattern. Section 6.3 takes up the first and Section 6.4 the second.

6.2.5 Summary

A convolutional layer cross-correlates its input with a small kernel and adds a bias: every output element is the sum of the elementwise product of the kernel with one window of the input, and the window visits every position where it fits. The same two weights $(1, -1)$ then find a vertical edge wherever it occurs, and gradient descent recovers these weights from an image and its edge map, as it learns the kernels of every layer of a CNN from the loss. Mathematical convolution flips the kernel, which makes no difference when the kernel is learned. The operation can be written as one matrix product, which accelerators execute efficiently. Stacking layers enlarges the region of the input that each unit sees, by $k - 1$ pixels per layer with a $k \times k$ kernel.

6.2.6 Exercises

1. code **Diagonal edges.** Construct a 6×6 image `X` whose entries are 1 on and below the main diagonal and 0 above it. Let $\mathbf{K}$ be the kernel of Equation 6.9.
    1. Apply $\mathbf{K}$ and $\mathbf{K}^\top$ to `X` with `corr2d`. Give the positions and signs of the nonzero entries of each output, and explain them from the pixel differences that each kernel computes.
    2. Let $\mathbf{A}$ be any input and $\mathbf{B}$ any kernel. Show that cross-correlating $\mathbf{A}^\top$ with $\mathbf{B}$ gives the transpose of cross-correlating $\mathbf{A}$ with $\mathbf{B}^\top$. What does this identity predict for `corr2d` applied to the transpose of `X` with $\mathbf{K}$?
2. code **Hand-designed kernels.** Design some kernels manually.
    1. Given a directional vector $\mathbf{v} = (v_1, v_2)$, derive an edge-detection kernel that detects edges orthogonal to $\mathbf{v}$, i.e., edges in the direction $(v_2, -v_1)$.
    2. Derive a finite difference operator for the second derivative. Which structures in images respond most strongly to it?
    3. What is the minimum size of a kernel to obtain a derivative of order d?
    4. How would you design a blur kernel? Why might you want to use such a kernel?
    5. Load a photograph, convert it to one channel, and downsample it to about 100×100 pixels (`corr2d` loops in Python). Apply the edge detector, the blur kernel, and the sharpening kernel "twice the identity minus the blur", and display each result next to the original. What does the edge map of a photograph show that the edge map of the black band could not?

    Adapted from Stanford CS231n, Assignment 2, ConvolutionalNetworks notebook.
3. code **Convolution as a banded matrix.** The patch matrix of Section 6.2.3 unfolds the input. The opposite construction unrolls the kernel. Let $\mathbf{X}$ be a 4×4 input, $\mathbf{K}$ a 3×3 kernel, and $\mathrm{vec}(\cdot)$ the row-by-row flattening.
    1. Construct the 4×16 matrix $\mathbf{W}$ with $\mathbf{W}\,\mathrm{vec}(\mathbf{X}) = \mathrm{vec}(\mathbf{Y})$, where $\mathbf{Y}$ is the cross-correlation of $\mathbf{X}$ with $\mathbf{K}$. How many nonzero entries does each row have, and how many distinct values appear in $\mathbf{W}$?
    2. Build $\mathbf{W}$ in code from `K` and confirm that the product reproduces `corr2d(X, K)`.

3. For an $n \times n$ input and a $k \times k$ kernel, count the entries of $\mathbf{W}$ stored as a dense array, the nonzero entries of $\mathbf{W}$, and the entries of the patch matrix. How does each count grow with n?
4. Keep the sparsity pattern of $\mathbf{W}$ but give every row its own k^2 values. The result is a locally connected layer: the fourth-order tensor `V` of Equation 6.2 restricted to a $k \times k$ window of offsets (a, b), before translation equivariance ties the weights. For an $n \times n$ input, count the parameters of this layer, of a fully connected layer with the same input and output shapes, and of the convolution.

Adapted from CMU 11-785, Homework 2 Part 1.

4. code **Effective receptive field.** ▣ The theoretical receptive field r of this section bounds which inputs *can* influence an output; the effective receptive field measures how much they actually do (Luo et al. 2016). Build a stack of L single-channel 3×3 convolutional layers with no padding, no bias, and no nonlinearity on a 32×32 input, using the built-in layer of Section 6.2.2.3. Draw every kernel entry independently from a standard normal distribution.

   1. For $L = 3$ and a single draw of the kernels, backpropagate a unit gradient from an output unit near the center of the last layer to the input and plot the magnitude of the input gradient. Compare its support with the 7×7 theoretical receptive field, and say where the magnitude is largest.
   2. Average the squared input gradient over 500 draws of the kernels, for $L = 3$ and for $L = 10$. Sum each averaged map over its rows and normalize the resulting column profile to sum to one. How does the ratio of the profile's standard deviation to the theoretical side length $2L + 1$ change with depth?
   3. Show that the expected squared input gradient, summed over rows, is proportional to the convolution of L copies of the vector $(1, 1, 1)$. Derive how the standard deviation of the normalized profile grows with L.

5. code **Backward pass of a convolution.** ▣ Let $\mathbf{Y}$ be the single-channel cross-correlation of an input $\mathbf{X}$ with a kernel $\mathbf{K}$, as computed by `corr2d`, and let ℓ be a scalar loss that depends on $\mathbf{X}$ and $\mathbf{K}$ only through $\mathbf{Y}$.

   1. Derive $\partial \ell / \partial \mathbf{K}$ and $\partial \ell / \partial \mathbf{X}$ in terms of $\partial \ell / \partial \mathbf{Y}$, in the style of the backpropagation equations of Section 4.3.
   2. Implement your expression for $\partial \ell / \partial \mathbf{K}$ without automatic differentiation for the loss $\ell = \sum_{i,j} G_{ij} Y_{ij}$, where $\mathbf{G}$ is a fixed random matrix of the shape of $\mathbf{Y}$. Compare it with a centered finite-difference gradient with step size 10^{-2} on a small random `X` and `K`, and report the largest relative error.
   3. Check your expression for $\partial \ell / \partial \mathbf{K}$ against the kernel gradient that automatic differentiation computes in the experiment of Section 6.2.2.3.

Adapted from Michael Nielsen, Neural Networks and Deep Learning, Chapter 6, and Stanford CS231n, Assignment 2, ConvolutionalNetworks notebook.

6.3 Padding, Stride, and Dilation

The convolutional layer of Section 6.2 evaluates its kernel only where the kernel fits inside the input, so its output is smaller than its input by one less than the kernel size along each axis. For a single layer this hardly matters. In a deep network the size of every output becomes a design decision, and two needs pull in opposite directions. How do we keep a

stack of layers from cutting away the border of the image? And how do we shrink the image on purpose, so that a unit near the top can see all of it without a network of hundreds of layers?

Ten layers with 5×5 kernels are enough to show the first problem. Each layer trims two pixels from every side of its input, so a 240×240 image comes out as 200×200. The stack has removed 30% of the image's area, all of it at the border. The second problem is the reverse. A network that never shrinks its input computes a full-resolution output in every layer, and its units see only a few pixels farther with each layer. Occasionally we want it both ways, a wide view and an output at every pixel, without the weights of a large kernel.

Three settings of a convolutional layer address these needs. *Padding* adds a border of extra pixels around the input, so that the kernel can be placed at the edge. *Stride* moves the window by more than one pixel at a time. *Dilation* spaces the kernel's weights apart. The formulas for the output shape, which we derive below, are a matter of counting window positions. The more interesting question is how far a unit in a deep network sees, and here the three settings differ a great deal. We stay with single-channel inputs, as in Section 6.2, and add channels in Section 6.4.

6.3.1 Padding

6.3.1.1 Boundary Pixels

Pixels near the border are underused even before a layer cuts them away. Figure 6.8 counts the windows that contain each pixel. With a $k \times k$ kernel, a pixel at least $k - 1$ pixels away from every border lies in k^2 windows, a pixel on an edge in at most k, and a corner pixel in a single one. A 5×5 kernel covers each interior pixel in 25 windows but a corner pixel in only one, so an object near the corner of an image barely reaches the next layer. Each layer thus draws most of its evidence from the center of its input.

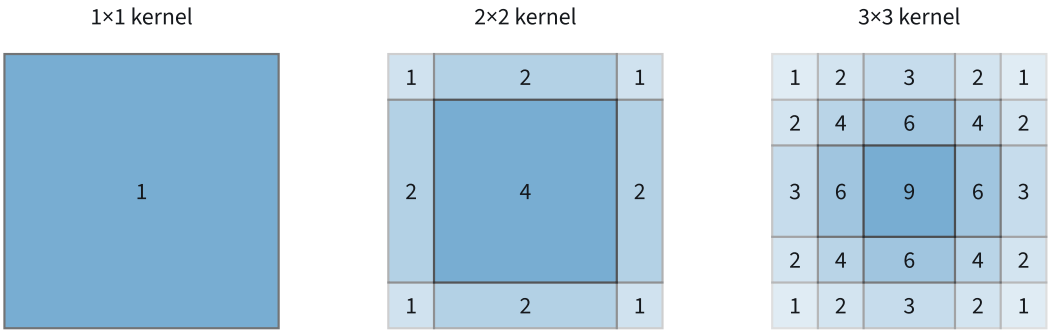


Figure 6.8: Number of windows that contain each pixel when a 1×1, 2×2, or 3×3 kernel slides over a large input without padding. Interior pixels are covered k^2 times, corner pixels once. Each stretched cell stands for all pixels at that distance from the border, however many there are.

6.3.1.2 Zero Padding

The remedy is to enlarge the input by a border of extra pixels before applying the kernel, usually pixels of value zero. This is called *padding*. In Figure 6.9, one row of zeros above

and below and one column on either side turn the 3×3 input into a 5×5 one, and the 2×2 kernel now produces a 4×4 output instead of a 2×2 one. The shaded first output element is computed entirely from padding and the input's corner: $0 \times 0 + 0 \times 1 + 0 \times 2 + 0 \times 3 = 0$.

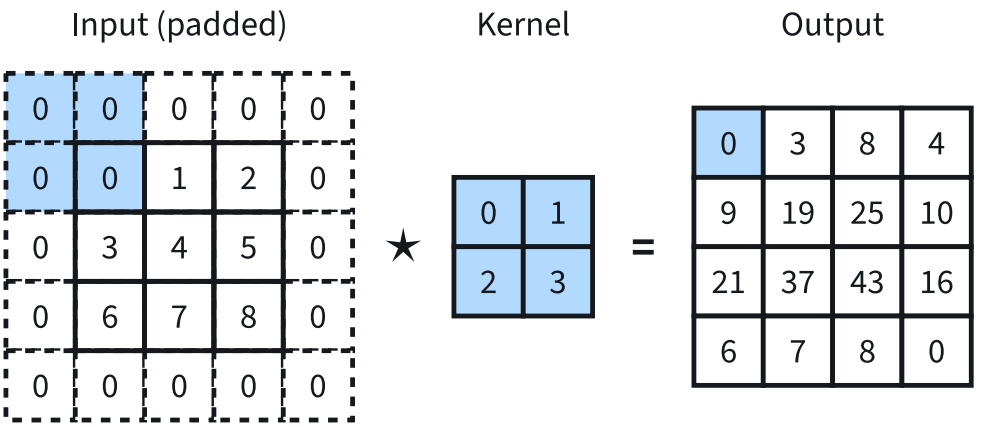


Figure 6.9: Two-dimensional cross-correlation with one pixel of zero padding on every side.

We count padding per side, as the code will: p_{h} rows are added above the input and p_{h} below it, and p_{w} columns on the left and on the right. The padded input then has shape $(n_{\mathrm{h}} + 2p_{\mathrm{h}}) \times (n_{\mathrm{w}} + 2p_{\mathrm{w}})$, and the output has shape

$$(n_{\mathrm{h}} + 2p_{\mathrm{h}} - k_{\mathrm{h}} + 1) \times (n_{\mathrm{w}} + 2p_{\mathrm{w}} - k_{\mathrm{w}} + 1). \tag{6.10}$$

Each row of padding adds one row to the output, and each column adds one column. The kernel removed $k_{\mathrm{h}} - 1$ rows, so padding $p_{\mathrm{h}} = (k_{\mathrm{h}} - 1)/2$ rows on each side, and likewise $p_{\mathrm{w}} = (k_{\mathrm{w}} - 1)/2$ columns, gives the output the shape of the input. This is the usual choice: a stack of such layers keeps its resolution however deep it is.

The division by two is the reason why CNNs almost always use odd kernels, of size 1, 3, 5, or 7 with padding 0, 1, 2, or 3. For an odd kernel the window that computes the output `Y[i, j]` is centered on the input `X[i, j]`, so the output is aligned with the input. An even kernel has $k_{\mathrm{h}} - 1$ odd, and keeping the shape then takes one more row on one side than on the other. The kernel has no center pixel, and the output is shifted by half a pixel.

To check the formula in code, we need a small helper. Convolutional layers expect a mini-batch of examples with several channels, so `comp_conv2d` adds two dimensions of size 1 to a two-dimensional input, applies the layer, and removes them again.

```
def comp_conv2d(conv2d, X):
    """Apply a convolutional layer to a single-channel 2D input."""
    Y = conv2d(X.reshape((1, 1) + X.shape))  # input (1, 1, h, w)
    return Y.reshape(Y.shape[2:])
```

A 3×3 kernel needs $p = (3 - 1)/2 = 1$, which is `padding=1` in code. On an 8×8 input it should give an 8×8 output.

```
X = torch.rand(size=(8, 8))
conv2d = nn.LazyConv2d(1, kernel_size=3, padding=1)
comp_conv2d(conv2d, X).shape
```

```
torch.Size([8, 8])
```

6.3.1.3 Padding Values

Zero padding treats the image as if it were black outside its frame. This is a *boundary condition*: a choice of values beyond the border, which the image itself does not provide. To see its effect, we cross-correlate a uniformly bright 5×5 image of ones, padded by one pixel of zeros on every side, with a 3×3 kernel of ones, using `corr2d` from Section 6.2. Each output then counts the image pixels in its window.

```
img = d2l.ones((5, 5))
d2l.corr2d(F.pad(img, (1, 1, 1, 1)), d2l.ones((3, 3)))
```

```
tensor([[4., 6., 6., 6., 4.],
        [6., 9., 9., 9., 6.],
        [6., 9., 9., 9., 6.],
        [6., 9., 9., 9., 6.],
        [4., 6., 6., 6., 4.]])
```

The interior outputs are 9, those along the edges 6, and those in the corners 4: the layer sees a dark frame around an image that has none. The frame travels inward through later layers, which can learn to detect it, and the resulting artifacts near the border can leave parts of the input to which the network barely responds (Alsallakh et al. 2020). Two other boundary conditions are common. *Reflect padding* mirrors the rows and columns next to the border, and *replicate padding* repeats the border pixels themselves. Both keep a uniform image uniform, and image-generation networks often use them because a dark frame would be visible in their output. Zero padding remains the default in classifiers: it is cheap, and libraries apply it without building a larger tensor.

6.3.2 Stride

6.3.2.1 Strided Convolution

A network that keeps a 224×224 image at full resolution computes 50,176 outputs per channel in every layer. Since each 3×3 layer widens the region a unit sees by only two pixels, such a network would need more than 100 layers before a single unit could see the whole image, as a classifier must. Moving the window in larger steps reduces both the number of outputs and the number of layers.

So far the window has moved one pixel at a time, to the right and then down. The number of rows and columns it moves per step is the *stride*. With stride s_h along the height and s_w along the width, the layer evaluates only every s_h-th row and every s_w-th column of window positions, so it computes about $s_\mathrm{h} s_\mathrm{w}$ times fewer outputs. Figure 6.10 uses stride 3 along the height and 2 along the width on the padded input of Figure 6.9, an unusual pair that keeps the two axes easy to tell apart. Each shaded output has the color of its window. The blue one is $0 \times 0 + 0 \times 1 + 1 \times 2 + 2 \times 3 = 8$, whose window lies two columns to the right of the first window, and the orange one is $0 \times 0 + 6 \times 1 + 0 \times 2 + 0 \times 3 = 6$, whose window lies

three rows below it. Moving two more columns to the right would put the window partly outside the padded input, so each row has two outputs.

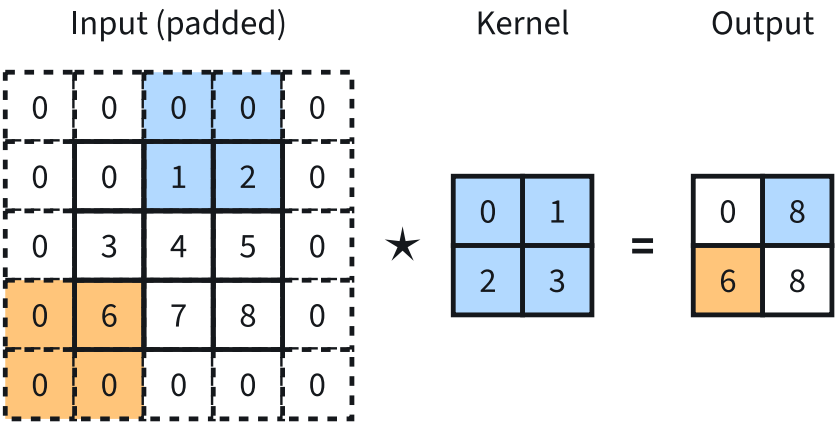


Figure 6.10: Cross-correlation with strides of 3 and 2 for height and width, respectively. Each shaded output has the color of the window that produces it.

To count the outputs along the height, note that the window starts in row 0 of the padded input and that its last row must stay inside the $n_\mathrm{h} + 2p_\mathrm{h}$ rows. The possible starting rows are $0, s_\mathrm{h}, 2s_\mathrm{h}, \ldots$, up to $n_\mathrm{h} + 2p_\mathrm{h} - k_\mathrm{h}$, so with the same count for the width the output shape is

$$\left(\left\lfloor \frac{n_\mathrm{h} + 2p_\mathrm{h} - k_\mathrm{h}}{s_\mathrm{h}} \right\rfloor + 1\right) \times \left(\left\lfloor \frac{n_\mathrm{w} + 2p_\mathrm{w} - k_\mathrm{w}}{s_\mathrm{w}} \right\rfloor + 1\right). \tag{6.11}$$

The fraction counts the steps the window can take and the $+1$ counts its starting position. For Figure 6.10 this gives $\lfloor 3/3 \rfloor + 1 = 2$ rows and $\lfloor 3/2 \rfloor + 1 = 2$ columns, and at stride 1 it is the padding formula. With shape-preserving padding, $2p_\mathrm{h} = k_\mathrm{h} - 1$, the number of rows is $\lfloor (n_\mathrm{h} - 1)/s_\mathrm{h} \rfloor + 1 = \lceil n_\mathrm{h}/s_\mathrm{h} \rceil$: stride s divides the resolution by s, rounding up.

The most common strided layer has a 3×3 kernel, padding 1, and stride 2. It halves an 8×8 input to 4×4.

```
conv2d = nn.LazyConv2d(1, kernel_size=3, padding=1, stride=2)
comp_conv2d(conv2d, X).shape
```

```
torch.Size([4, 4])
```

Height and width may use different kernel sizes, paddings, and strides; the formula applies to each axis separately.

When the stride equals the kernel size and there is no padding, the windows tile the input without overlap. A 16×16 kernel with stride 16 cuts a 224×224 image into a 14×14 grid of patches and maps each patch to one output per channel. This *patchify* layer is how vision transformers turn an image into a sequence of patches, and some convolutional networks begin the same way.

When height and width use the same setting, we drop the subscripts and write p and s. A layer that is given neither setting uses stride 1 and no padding.

6.3.2.2 Stride and the Receptive Field

We saw in Section 6.2 that at stride 1 each $k \times k$ layer widens the receptive field by $k-1$ pixels along each axis, so that it grows only linearly with depth. Stride changes this. After a layer with stride 2, neighboring positions of its output lie two input pixels apart, and a 3×3 kernel applied to that output reaches $(3-1) \cdot 2 = 4$ input pixels further instead of 2. In general, one step in the input of layer i spans $\prod_{j<i} s_j$ pixels of the original input, and the top unit of a stack of L layers with kernel sizes k_i and strides s_i has a receptive field of side

$$r = 1 + \sum_{i=1}^{L} (k_i - 1) \prod_{j=1}^{i-1} s_j, \tag{6.12}$$

where the empty product for $i = 1$ equals 1. Each layer adds the extent of its kernel, measured in pixels of the original input; at stride 1 every term is $k_i - 1$, and we recover the count of Section 6.2. Padding does not appear, because it changes which outputs exist, not how far each one sees. For three 3×3 layers, stride 1 everywhere gives $r = 1 + 2 + 2 + 2 = 7$, while strides 2, 2, 1 give $r = 1 + 2 + 4 + 8 = 15$.

Every stride multiplies the reach of all later kernels, so the receptive field grows geometrically for as long as the network keeps striding. After five halvings, common in classifiers for 224×224 images, a kernel reaches 32 times further than it would at the bottom, and no unit needs hundreds of layers to see the whole image.

6.3.3 Dilation

6.3.3.1 Dilated Convolution

Stride lets a unit see far by giving up resolution. Some tasks need both: a wide view of the image and an output at every pixel. A large kernel would give the wide view, but its weights and its computation grow with the square of its size. Padding and stride change where the kernel is applied. Dilation changes the kernel's footprint itself: a *dilated convolution* (Yu and Koltun 2016) places the kernel's weights d pixels apart instead of at adjacent pixels, where d is the *dilation*. A $k \times k$ kernel with dilation d still has k^2 weights, but along each axis they now span

$$k' = k + (k-1)(d-1) = d(k-1) + 1$$

pixels, the *effective kernel size*. Along each axis, $d-1$ pixels are skipped between neighboring weights, and $d = 1$, the default, is the ordinary convolution. Figure 6.11 shows a 3×3 kernel at dilations 1 and 2. At dilation 2 the nine weights spread over a 5×5 footprint, skipping every other pixel, while each output still takes nine multiplications.

For output shapes there is nothing new to learn: a dilated kernel behaves like a dense kernel of its effective size. Substituting $k' = d(k-1) + 1$ for k in the strided formula gives the number of outputs along one axis for padding, stride, and dilation together,

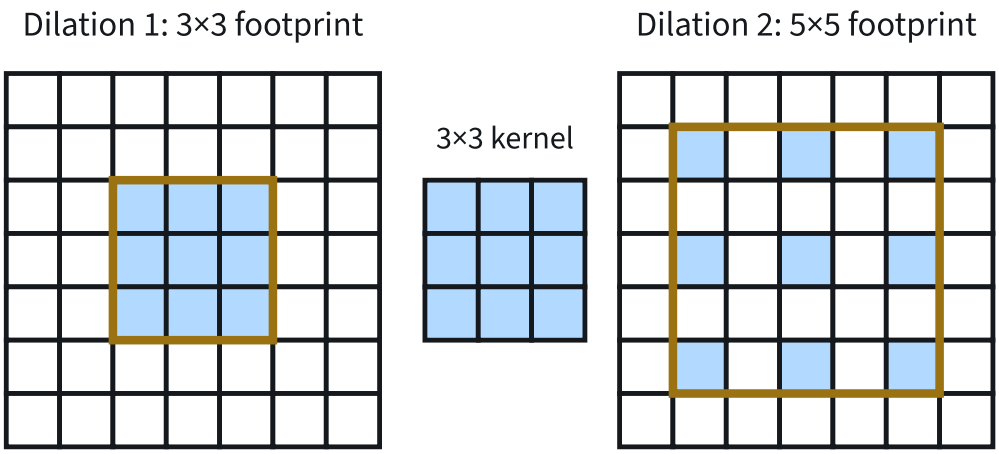


Figure 6.11: A 3×3 kernel at dilation 1 and dilation 2. At dilation 2 its nine weights span a 5×5 footprint.

$$\left\lfloor \frac{n + 2p - d(k-1) - 1}{s} \right\rfloor + 1, \tag{6.13}$$

and the same holds for the other axis with its own n, k, p, s, and d. With dilation 1 this is the strided formula, and with stride 1 and no padding it is the $n - k + 1$ rule we started from. On our 8×8 input, a 3×3 kernel at dilation 2 and a dense 5×5 kernel should both give a 4×4 output.

```
conv2d = nn.LazyConv2d(1, kernel_size=3, dilation=2)
conv5 = nn.LazyConv2d(1, kernel_size=5)
comp_conv2d(conv2d, X).shape, comp_conv2d(conv5, X).shape
```

```
(torch.Size([4, 4]), torch.Size([4, 4]))
```

6.3.3.2 Dilation and the Receptive Field

A dilated layer at stride 1 widens the receptive field by $k' - 1 = d(k-1)$ pixels, so Equation 6.12 holds with each k_i replaced by its effective size. Doubling the dilation from layer to layer, $d_i = 1, 2, 4, \ldots, 2^{L-1}$, gives L layers of 3×3 kernels a receptive field of side $1 + 2(1 + 2 + \cdots + 2^{L-1}) = 2^{L+1} - 1$. It grows exponentially with depth, like that of a strided stack, but every layer keeps the resolution of the input.

Figure 6.12 compares the two ways to reach far. Panel (a) is the strided stack computed above: with strides 2, 2, 1 the three layers add 2, 4, and 8 pixels to the single pixel at the top, 15 in all. Panel (b) reaches the same 15 pixels with dilations 1, 2, 4, and the top unit depends on the input through the same connections. The strided stack keeps only every fourth position; the dilated stack computes all of them. Without padding and with the same weights, the strided stack returns every fourth output of the dilated stack, even with nonlinearities between the layers. In other words, the dilated stack is the strided stack with nothing discarded. This matters for *dense prediction* tasks such as semantic segmentation,

which assigns a label to every pixel and thus needs both wide context and full resolution, and dilated convolutions are widely used there (Yu and Koltun 2016). Their cost is that every layer computes its outputs at full resolution. A network that downsamples with strides can instead restore the resolution at its end with learned upsampling.

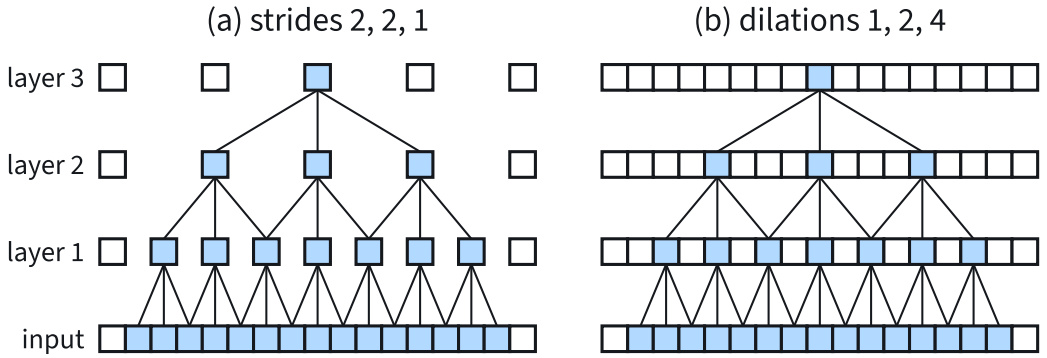


Figure 6.12: Receptive field of the top unit of three stacked layers with kernel size 3, in one dimension. Cells are the positions each layer computes, shaded cells those the top unit depends on, and lines the kernel's taps. (a) Strides 2, 2, 1: layer 1 keeps every second position, layers 2 and 3 every fourth. (b) Stride 1 with dilations 1, 2, 4: every layer keeps all positions. Both top units see the same 15 input pixels.

Throughout this section the input had a single channel and the layer a single kernel. Real images have several channels, and a useful layer detects more than one pattern. Section 6.4 removes both restrictions.

6.3.4 Summary

A convolution shrinks its input by $k-1$ pixels along each axis and uses border pixels in fewer windows than interior ones. Padding $(k-1)/2$ pixels on each side of an odd kernel keeps the shape and centers each output on its input pixel; zero padding imposes a boundary condition that the network can detect. Stride s evaluates every s-th window, divides the resolution and the cost by about s along each axis, and makes the receptive field of later layers grow geometrically. Dilation d spreads a kernel's weights to an effective size $d(k-1)+1$, which enlarges the receptive field at full resolution and at constant cost per output. One formula, Equation 6.13, gives the output shape for all three.

6.3.5 Exercises

1. code **Mirror padding.** Mirror padding, also called reflect padding, extends a tensor by reflecting its entries about the border without repeating the border entry: padding the vector $(1,2,3)$ by one entry on each side gives $(2,1,2,3,2)$.
    1. Implement mirror padding by one entry on every side of a two-dimensional tensor, and check the result by hand on a 3×3 tensor with distinct entries.
    2. Repeat the computation of Section 6.3.1.3 with mirror padding instead of zero padding, and explain the result.
    3. Take the 5×5 ramp $x_{ij}=j$ and the 3×3 kernel whose entries all equal $\frac{1}{9}$. Show that in the interior the output equals the input. Compute the outputs in the first

and last columns under zero padding and under mirror padding, and find a way to pad that reproduces the ramp exactly.

2. **Benefits of stride.** A 3×3 convolution with one pixel of padding on every side is applied to a single-channel 64×64 input.

   1. Count the multiplications this layer performs at stride 1 and at stride 2. By what factor does stride 2 reduce the cost of a second, stride-1 layer of the same kind placed after this one?
   2. Four such layers are stacked. Use Equation 6.12 to compute the receptive field of one output element when all four layers have stride 1 and when the first two have stride 2. An object in the image is 20 pixels wide. Which of the two stacks has units that can see all of it at once?

3. **Stride as sampling.** For a length-6 one-dimensional input, write the 4×6 weight matrix of a stride-1, kernel-3 convolution in the banded-matrix form of Section 6.2.3, and the 2×4 zero–one matrix that keeps every other entry of a length-4 vector. Show that composing the two reproduces the matrix of the stride-2, kernel-3 convolution.

   Adapted from Simon Prince, Understanding Deep Learning, Problem 10.15.

4. **Dilation and gridding.** A network stacks four 3×3 convolutions with stride 1. The doubling dilation schedule $(1, 2, 4, 8)$ gives one output element a receptive field of side 31. Compare it with the constant schedule $(4, 4, 4, 4)$.

   1. Use Equation 6.12, with each kernel replaced by its effective size, to compute the receptive field of the constant schedule.
   2. For each schedule, determine which pixels inside the receptive field the output actually depends on, and count them. A schedule whose output depends only on a sparse lattice within its receptive field is said to exhibit *gridding*.
   3. Give one reason gridding is harmful for dense prediction tasks such as segmentation.
   4. ▣ State a condition on the dilations $d_1, \dots, d_L$ of a stack of 3×3 convolutions with stride 1 that guarantees that the output depends on every pixel of its receptive field, and show that the condition suffices.

5. **Strides versus dilations.** Consider the two one-dimensional stacks of Figure 6.12, without padding, with kernels $\mathbf{w}_1, \mathbf{w}_2, \mathbf{w}_3$ of size 3 shared by both stacks and an elementwise nonlinearity after every layer.

   1. Write the output of each stack at position i as a function of the input, and show that the output of the strided stack at position i equals the output of the dilated stack at position $4i$.
   2. For an input of length n, count the multiplications each stack performs, ignoring the border.
   3. Generalize the first part: which dilations reproduce a stack with strides $s_1, \dots, s_L$?

6. **Fractional stride.** ▣ Read a stride of $\frac{1}{2}$ as moving the convolution window by half an input element per output element.

   1. Evaluate Equation 6.13 at $d = 1$, $s = \frac{1}{2}$, and $p = (k-1)/2$ for an odd kernel size k. How does the output height compare with the input height?
   2. Describe how to compute this output with an ordinary stride-1 convolution applied to a modified input, and check that your construction gives the same output

height.
3. Name a task in which a network needs this operation, and say where in the network it would be placed.

7. **Shapes through a stem.** A 224×224 image passes through four layers: a 7×7 convolution with padding 3 and stride 2, a 3×3 convolution with padding 1 and stride 2, and two 3×3 convolutions with padding 1 and stride 1.
   1. Give the height of the output of every layer.
   2. Use Equation 6.12 to compute the receptive field of a unit of the last layer. What fraction of the image's width does it see?
   3. Replace the first two layers by a single 4×4 convolution with stride 4 and no padding, the patchify layer of this section. Give the output height and the receptive field again. Do neighboring windows of the first layer still overlap?

6.4 Multiple Input and Multiple Output Channels

Every kernel we have used so far has read one grid of numbers and produced one. The edge detector of Section 6.2 looked at a grayscale image and found one kind of edge. Real images and real networks need more on both sides. A color photograph stores three numbers per pixel, one *channel* each for red, green, and blue, so a 224×224 RGB image is a tensor of shape $3 \times 224 \times 224$. A boundary between a red and a green region of equal brightness is visible only when the channels are compared. And a useful layer detects dozens of patterns at every position (edges at several orientations, spots, textures), each in a grid of its own. How does one layer read several input channels and produce several output channels, and what does that cost?

The convolutional layer of Section 6.1 already allows for this: each output channel adds up, over all input channels, a separate $k_\text{h} \times k_\text{w}$ kernel applied to each. This section implements that layer on top of the single-channel `corr2d` and counts what it costs. The count turns out to be governed by one quantity, the product $c_\text{i} c_\text{o}$ of the two channel counts, and the rest of the section asks how to make that product smaller.

6.4.1 Multiple Input Channels

Let us start with the input side. With c_i input channels, the kernel needs one $k_\text{h} \times k_\text{w}$ grid of weights for every input channel, so it becomes a tensor of shape $c_\text{i} \times k_\text{h} \times k_\text{w}$. We cross-correlate each input channel with the matching kernel channel and add the c_i results. Writing $\mathbf{X}_c$ and $\mathbf{K}_c$ for channel c of the input and of the kernel, and $\mathbf{X}_c \star \mathbf{K}_c$ for their two-dimensional cross-correlation Equation 6.7, which `corr2d` computes, the output is the single grid

$$\mathbf{Y} = \sum_{c=1}^{c_\text{i}} \mathbf{X}_c \star \mathbf{K}_c. \tag{6.14}$$

Each output value is thus a weighted sum over the same window in every channel, with $c_\text{i} k_\text{h} k_\text{w}$ products. Figure 6.13 works an example with two input channels. The shaded first output element is $(1 \times 1 + 2 \times 2 + 4 \times 3 + 5 \times 4) + (0 \times 0 + 1 \times 1 + 3 \times 2 + 4 \times 3) = 56$.

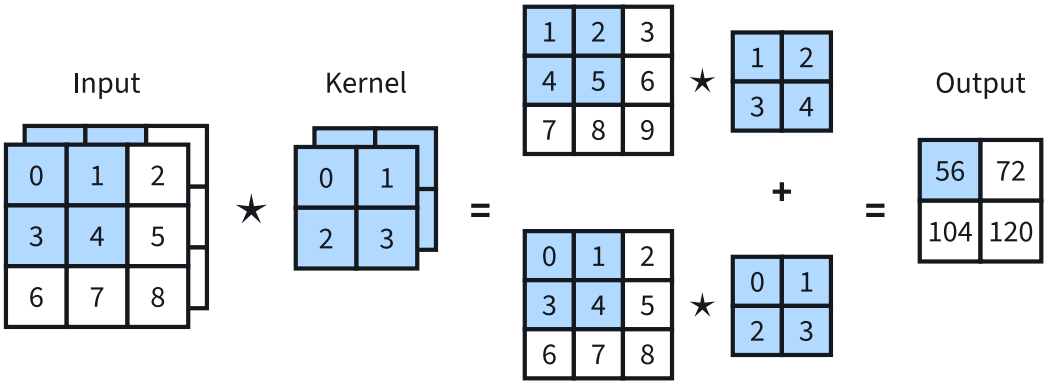


Figure 6.13: Cross-correlation with two input channels: one cross-correlation per channel, then a sum.

Why add the channels? Because the sum is what lets a kernel compare colors. A 1×1 kernel with weight $+1$ on the red channel and -1 on the green channel outputs red minus green at every pixel: positive in red regions, negative in green ones, and zero in gray ones, however bright. Where its sign flips between neighboring pixels, a red region meets a green one. This holds even when both regions are equally bright, so that a filter applied to the brightness alone sees no edge there.

The implementation is the formula read aloud: one call of `corr2d` per channel, then a sum.

```
def corr2d_multi_in(X, K):
    """Cross-correlate each channel of X with its channel of K and add."""
    return sum(d2l.corr2d(x, k) for x, k in zip(X, K))
```

With the input and kernel of Figure 6.13, the first output element should be 56.

```
X = d2l.tensor([[[0.0, 1.0, 2.0], [3.0, 4.0, 5.0], [6.0, 7.0, 8.0]],
                [[1.0, 2.0, 3.0], [4.0, 5.0, 6.0], [7.0, 8.0, 9.0]]])
K = d2l.tensor([[[0.0, 1.0], [2.0, 3.0]], [[1.0, 2.0], [3.0, 4.0]]])

corr2d_multi_in(X, K)
```

```
tensor([[ 56.,  72.],
        [104., 120.]])
```

6.4.2 Multiple Output Channels

A multi-input kernel, however many channels it reads, produces a single feature map (Section 6.1.4): the responses of one detector. To detect c_{o} patterns, the layer simply uses c_{o} such kernels, one per output channel, each of shape $c_{\mathrm{i}} \times k_{\mathrm{h}} \times k_{\mathrm{w}}$. Stacked, they form a kernel tensor of shape $c_{\mathrm{o}} \times c_{\mathrm{i}} \times k_{\mathrm{h}} \times k_{\mathrm{w}}$, and output channel o is

$$\mathbf{Y}_o = \sum_{c=1}^{c_\mathrm{i}} \mathbf{X}_c \star \mathbf{K}_{o,c}, \quad o = 1, \ldots, c_\mathrm{o}. \tag{6.15}$$

Every output channel reads every input channel, each through a kernel of its own. It is worth remembering this dense wiring, since the cost of the layer comes from it. This is the layer Equation 6.6 of Section 6.1, written channel by channel, with $\mathbf{K}_{o,c}$ its $k_\mathrm{h} \times k_\mathrm{w}$ slice for output channel o and input channel c. Networks usually increase the number of channels as strides or pooling reduce the resolution: they give up spatial detail and gain more kinds of features at each position.

PyTorch stores the kernel in this order, channels first, as it does its inputs.

The implementation applies `corr2d_multi_in` once per output channel and stacks the results.

```
def corr2d_multi_in_out(X, K):
    """Apply corr2d_multi_in once per output channel of K and stack."""
    return d2l.stack([corr2d_multi_in(X, k) for k in K], 0)
```

To test it, we build a kernel with three output channels from `K`, `K + 1`, and `K + 2`.

```
K = d2l.stack((K, K + 1, K + 2), 0)
K.shape
```

```
torch.Size([3, 2, 2, 2])
```

```
corr2d_multi_in_out(X, K)
```

```
tensor([[[ 56.,  72.],
         [104., 120.]],

        [[ 76., 100.],
         [148., 172.]],

        [[ 96., 128.],
         [192., 224.]]])
```

The first channel reproduces the output above, because its kernel is the original `K`. The second can be checked in the head. Adding 1 to every weight adds to each output the sum of its window over both input channels, which is 20 for the first element, so the second channel starts at $56 + 20 = 76$.

6.4.2.1 Parameters and Cost

Channels multiply the size of a layer. Before we design networks with many of them, we should know by how much. With one bias per output channel, a convolutional layer has

$$c_\mathrm{o} c_\mathrm{i} k_\mathrm{h} k_\mathrm{w} + c_\mathrm{o} \tag{6.16}$$

parameters. For the first layer of a network this is a small number: the layer from 3 to 64 channels with 3×3 kernels in Section 6.1 has 1,792 of them. A layer deeper in, from 256 to 256 channels with 3×3 kernels, has 589,824 weights and 256 biases.

What does it cost to apply the layer? Each output value takes one multiply-add per weight of its output channel, $c_\mathrm{i} k_\mathrm{h} k_\mathrm{w}$ in total, so a layer with an $h_\mathrm{o} \times w_\mathrm{o}$ output performs

$$h_\mathrm{o} w_\mathrm{o} \cdot c_\mathrm{o} c_\mathrm{i} k_\mathrm{h} k_\mathrm{w} \tag{6.17}$$

multiply-adds. On a 56×56 output, the 256-channel layer performs $56^2 \cdot 589{,}824 \approx 1.8 \times 10^9$ of them for a single image. Both counts grow with the product $c_\mathrm{i} c_\mathrm{o}$: doubling the number of channels in every layer quadruples them. The computation grows, in addition, with the number of output positions, since the same weights are applied at each.

There is a third cost, and it is easy to overlook. Training keeps each layer's output, $h_\mathrm{o} w_\mathrm{o} c_\mathrm{o}$ values per example, for the backward pass (Section 5.7.3.2). This activation memory is largest in the early layers, where the resolution is high and the weights are few. The 3-to-64-channel layer with a 224×224 output keeps $224^2 \cdot 64 \approx 3.2 \times 10^6$ values per example, against its 1,792 parameters.

6.4.3 The 1×1 Convolution

At first sight, a 1×1 kernel, $k_\mathrm{h} = k_\mathrm{w} = 1$, is a convolution in name only: it looks at a single pixel and cannot detect spatial patterns. With several channels it still computes something useful, as the red-minus-green kernel above did. At every position it maps the c_i input values to c_o output values with the same $c_\mathrm{o} \times c_\mathrm{i}$ matrix of weights: it is a fully connected layer applied to each pixel separately, with weights shared across pixels. Figure 6.14 shows the case of 3 input and 2 output channels; height and width are unchanged.

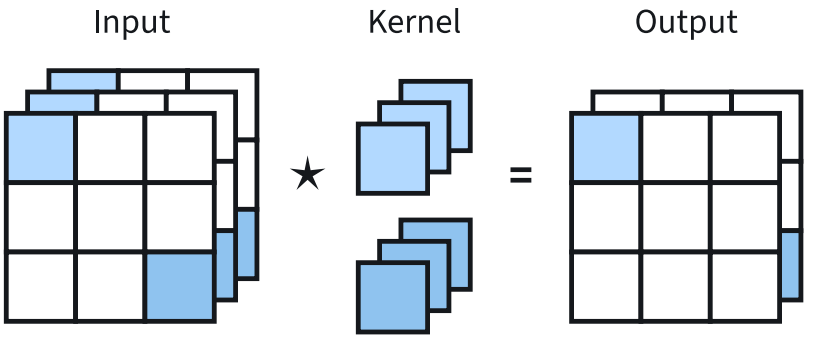


Figure 6.14: A 1×1 convolution with three input channels and two output channels. Each output element is a linear combination of the input elements at the same position.

Because the same matrix acts at every pixel, the whole layer is one matrix product. We flatten the spatial axes of the input into a $c_\mathrm{i} \times hw$ matrix, multiply it by the kernel reshaped to $c_\mathrm{o} \times c_\mathrm{i}$, and restore the spatial axes.

```
def corr2d_multi_in_out_1x1(X, K):
    """A 1x1 convolution as one matrix product."""
    c_i, h, w = X.shape
    c_o = K.shape[0]
    Y = d2l.matmul(d2l.reshape(K, (c_o, c_i)), d2l.reshape(X, (c_i, h * w)))
    return d2l.reshape(Y, (c_o, h, w))
```

We compare the matrix product with `corr2d_multi_in_out` on random data.

```
X = d2l.normal(0, 1, (3, 3, 3))
K = d2l.normal(0, 1, (2, 3, 1, 1))
Y1 = corr2d_multi_in_out_1x1(X, K)
Y2 = corr2d_multi_in_out(X, K)
float(d2l.reduce_sum(d2l.abs(Y1 - Y2)))
```

```
0.0
```

The two computations agree up to rounding error.

What is such a layer good for? It changes the number of channels cheaply, and deep networks use this to reduce the cost of the spatial convolutions around it. Take the 3×3 layer on 256 channels from above. A 1×1 convolution first reduces 256 channels to 64, the 3×3 convolution then works on 64 channels, and a second 1×1 convolution restores 256. The three layers have $16{,}384 + 36{,}864 + 16{,}384 = 69{,}632$ weights against $589{,}824$, about 8.5 times fewer, and the multiply-adds fall by the same factor. Of course, such a *bottleneck* can only represent maps that pass through 64 channels, a restriction that networks such as ResNet (Section 7.4) accept for the savings. A nonlinearity after a 1×1 convolution makes it a small nonlinear network applied at every pixel (Lin et al. 2013). Without one, it can be folded into the next convolution, since two linear maps compose to one.

Two further ways to cut the cost restrict which input channels each output channel reads: grouped convolutions, which ResNeXt uses (Section 7.4.5.1), and depthwise-separable convolutions, the building block of efficient networks (Section 7.5.1.1). We introduce each where a network first needs it.

All of these layers have one property that channels do nothing about: the output still shifts with the input, pixel by pixel. Section 6.5 summarizes neighborhoods so that small shifts of the input change the output little.

6.4.4 Summary

With several input channels, a convolution cross-correlates each input channel with its own kernel and adds the results, which lets one kernel combine evidence across channels such as colors. With several output channels, the layer holds one such kernel per output channel, a tensor of shape $c_\mathrm{o}\times c_\mathrm{i}\times k_\mathrm{h}\times k_\mathrm{w}$ with $c_\mathrm{o}c_\mathrm{i}k_\mathrm{h}k_\mathrm{w}+c_\mathrm{o}$ parameters and $h_\mathrm{o}w_\mathrm{o}c_\mathrm{o}c_\mathrm{i}k_\mathrm{h}k_\mathrm{w}$ multiply-adds, so the cost grows with the product of the channel counts. A 1×1 convolution is a fully connected layer applied at every pixel and changes the number of channels cheaply, as in bottlenecks.

6.4.5 Exercises

1. **Composition of convolutions.** Two convolutional layers with stride 1, no padding, and no nonlinearity in between have kernels of shape $c_{\mathrm{m}} \times c_{\mathrm{i}} \times k_1 \times k_1$ and $c_{\mathrm{o}} \times c_{\mathrm{m}} \times k_2 \times k_2$.
    1. Prove that their composition is a single convolution, and give the shape of its kernel.
    2. Compare the number of weights of the pair with that of the single convolution. For which intermediate width c_{m} is the pair cheaper?
    3. As Section 6.2 showed, three stacked single-channel 3×3 kernels have a 7×7 receptive field. Decide whether every single-channel 7×7 kernel is the composition of three 3×3 kernels, and justify the answer.
2. **Three costs, three layers.** A network for 224×224 RGB images begins with four 3×3 convolutions with padding 1: from 3 to 64 channels at stride 1, from 64 to 128 at stride 2, from 128 to 128 at stride 1, and from 128 to 256 at stride 2.
    1. For each layer give the output shape, the number of weights, the number of multiply-adds per image, and the number of output values that training must keep for the backward pass.
    2. Which layer has the most weights, which performs the most multiply-adds, and which keeps the most output values? Explain why these are three different layers.
    3. The images are replaced by 448×448 ones. Which of the three costs change, and by what factor?
3. code **Multi-channel im2col.** Extend the patch matrix of Section 6.2.3 to several input and output channels. Rebuild the two-channel `X` of Figure 6.13 and the three-output-channel `K` of Section 6.4.2, since later cells of this section overwrite both. Build the patch matrix, whose rows are input patches of shape $c_{\mathrm{i}} \times k_{\mathrm{h}} \times k_{\mathrm{w}}$ flattened in that order, and arrange `K` into a $c_{\mathrm{i}}k_{\mathrm{h}}k_{\mathrm{w}} \times c_{\mathrm{o}}$ kernel matrix whose rows follow the same order. Confirm that their product, rearranged into the shape of the output, reproduces `corr2d_multi_in_out(X, K)`.

    Adapted from Simon Prince, Understanding Deep Learning, Problems 10.6 and 10.7.

6.5 Pooling

The layers of the previous sections answer local questions. The edge detector of Section 6.2 reports, position by position, whether an edge lies there. A classifier must answer a question about the whole image: does this Fashion-MNIST picture show a shirt or a sneaker? This section asks how a network gets from many local answers to one global answer. Two things are needed.

The first is reach: some unit near the top of the network must have a receptive field (Section 6.2) that covers the whole image. At full resolution this takes patience, namely fourteen 3×3 layers for a 28×28 Fashion-MNIST image. We already know the remedy, which is to reduce the resolution on the way up (Section 6.3.2.2). With two halvings, six such layers are enough.

The second is new. The edge detector responds at the pixel where the edge lies. Shift the image by one pixel, as happens whenever the object or the camera moves slightly, and the response moves by one pixel too. A classifier's answer should not change because of such a shift.

A *pooling layer* halves the resolution as a stride does, but it also decides what to keep. It replaces each small window of its input by one summary value, the maximum or the average. The maximum of a window does not change when a response moves within it, and this is what bears on the second problem. How much it helps is something we can measure, and the answer is less than the word "invariance", often used of pooling, suggests. Invariance proper takes a window as large as the feature map, and even then the border has a say.

6.5.1 Maximum Pooling and Average Pooling

A pooling operator slides a window of fixed shape, the *pooling window*, over its input, like the kernel of a convolution. At each position it computes one output from the entries under the window. Unlike a convolution, it has no kernel and therefore nothing to learn: *maximum pooling* (*max-pooling* for short) outputs the largest entry in the window, and *average pooling* outputs their mean.

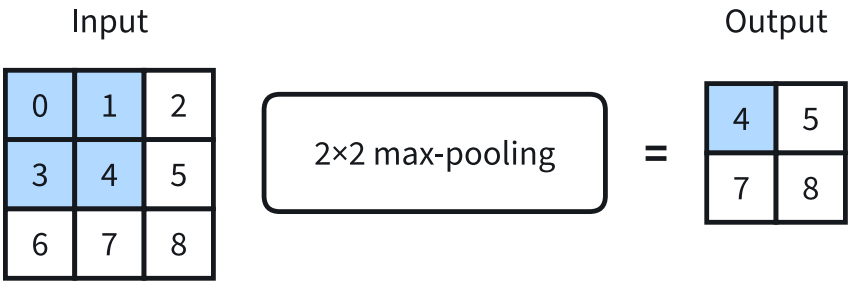


Figure 6.15: Max-pooling with a pooling window shape of 2×2. The shaded portions are the first output element as well as the input tensor elements used for the output computation: $\max(0, 1, 3, 4) = 4$.

In Figure 6.15 a 2×2 window slides over a 3×3 input with stride 1, from the upper left, left to right and top to bottom. It fits at two positions along each axis, so the output is 2×2, and each output entry is the maximum of the four entries under the window:

$$\begin{aligned}\max(0, 1, 3, 4) &= 4,\\ \max(1, 2, 4, 5) &= 5,\\ \max(3, 4, 6, 7) &= 7,\\ \max(4, 5, 7, 8) &= 8.\end{aligned}$$

Note that five of the nine entries, namely 0, 1, 2, 3, and 6, are the largest in no window. Small changes to them leave the output unchanged: a maximum takes no notice of the entries that lose. A pooling layer with a $k_\mathrm{h} \times k_\mathrm{w}$ window works the same way.

The function `pool2d` implements the forward computation. It has the loop structure of `corr2d` in Section 6.2, with the multiply-and-sum over the window replaced by a maximum or a mean. The argument `stride` moves the window by more than one position at a time; we need it below.

```
def pool2d(X, pool_size, mode='max', stride=1):
    k_h, k_w = pool_size
    Y = d2l.zeros(((X.shape[0] - k_h) // stride + 1,
                   (X.shape[1] - k_w) // stride + 1))
    for i in range(Y.shape[0]):
        for j in range(Y.shape[1]):
            window = X[i * stride: i * stride + k_h,
                       j * stride: j * stride + k_w]
            Y[i, j] = window.max() if mode == 'max' else window.mean()
    return Y
```

Applied to the input of Figure 6.15, `pool2d` should return 4, 5, 7, and 8.

```
X = d2l.tensor([[0.0, 1.0, 2.0], [3.0, 4.0, 5.0], [6.0, 7.0, 8.0]])
pool2d(X, (2, 2))
```

```
tensor([[4., 5.],
        [7., 8.]])
```

Average pooling of the same input returns the four window means, 2, 3, 5, and 6.

```
pool2d(X, (2, 2), 'avg')
```

```
tensor([[2., 3.],
        [5., 6.]])
```

6.5.2 Tolerance to Small Shifts

How much of a shift does pooling absorb? Let us try the simplest case there is. Pooling with a window of 2×2 and stride 2 splits its input into disjoint 2×2 blocks and reduces each block to one number. Consider a feature map with a single response of value 1, such as the output of an edge detector at an isolated edge, and move the response by one pixel at a time. We compute the max-pooled and the average-pooled outputs for three positions of the response in row 1 of a 4×4 map.

```
def single_response(i, j, n=4):
    """An n x n feature map that is 1 at (i, j) and 0 elsewhere."""
    return d2l.tensor([[float(r == i and c == j) for c in range(n)]
                       for r in range(n)])

for j in range(3):
    Z = single_response(1, j)
    print(f'response at (1, {j}): max',
          d2l.numpy(pool2d(Z, (2, 2), stride=2)).tolist(),
          'avg', d2l.numpy(pool2d(Z, (2, 2), 'avg', stride=2)).tolist())
```

```
response at (1, 0): max [[1.0, 0.0], [0.0, 0.0]] avg [[0.25, 0.0], [0.0, 0.0]]
response at (1, 1): max [[1.0, 0.0], [0.0, 0.0]] avg [[0.25, 0.0], [0.0, 0.0]]
response at (1, 2): max [[0.0, 1.0], [0.0, 0.0]] avg [[0.0, 0.25], [0.0, 0.0]]
```

Moving the response from column 0 to column 1 changes neither output: both positions lie in the same block. Moving it on to column 2 crosses into the next block, and the output moves with it. A one-pixel shift of a single response is therefore absorbed when the response stays inside its block, which happens for half of all horizontal positions, and it moves the pooled output by one position otherwise. Pooling makes a representation *tolerant* to some small shifts. It does not make it invariant to them. Figure 6.16 draws the max-pooled outputs for all four positions of the response.

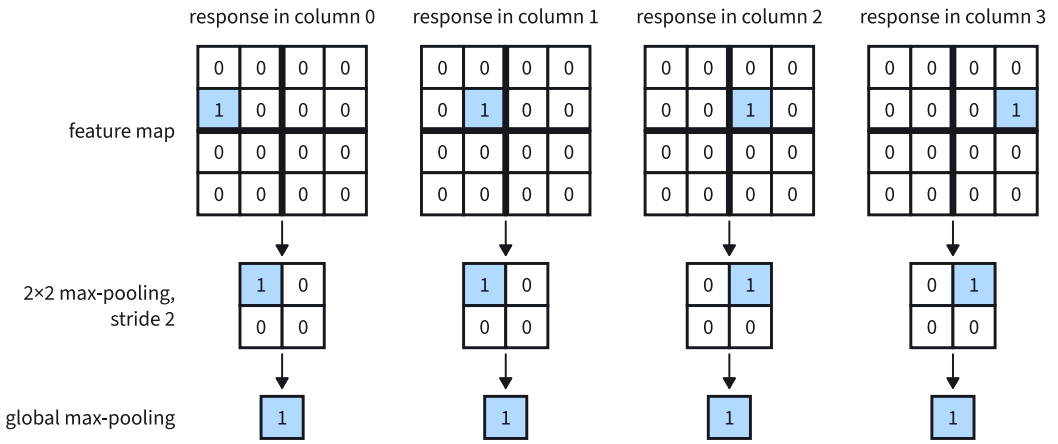


Figure 6.16: A single response moved through the four columns of a 4×4 feature map (top), its 2×2 max-pooled map with stride 2 (middle), and its global maximum (bottom). Heavy lines mark the pooling blocks. The pooled response stays put while the response stays inside its block and jumps when it crosses into the next one; the global maximum never moves.

The two kinds of pooling differ in what they report. Max-pooling reports the response at full strength, 1, wherever it falls in the block: it answers whether the feature is present somewhere in the window. Average pooling reports 0.25, because the response covers one of the four positions: it measures how much of the window the feature fills, so a strong but isolated response is diluted by its neighbors.

A stride of 2 keeps one value per block, so the output can jump under a one-pixel shift, as at the block boundary above. In the extreme case, keeping every second pixel of a pattern that alternates from pixel to pixel, $1, 0, 1, 0, \ldots$, gives all ones or all zeros, depending on a shift by one pixel. This error, which arises when a signal changes faster than the samples can follow, is called *aliasing*. It affects every stride-2 downsampler, pooled or convolutional. Smoothing the input before subsampling reduces it (Zhang 2019), and an exercise below works it out.

Tolerance to shifts is what this operation was invented for. The Neocognitron (Fukushima 1980) followed each layer of feature detectors by a layer of cells that responded when a detector fired anywhere in a small neighborhood. An early form of max-pooling was used in speech recognition (Yamaguchi et al. 1990), and the operation was later proposed as a model of how the visual cortex aggregates evidence for object recognition (Riesenhuber and Poggio 1999).

6.5.3 Global Pooling

What, then, about the invariance that Section 6.1 asked of the final answer, whether Waldo is anywhere in the scene? A window of fixed size cannot deliver it, as we have just seen. A window that covers the whole feature map can: the maximum over all positions, or the mean, is the same wherever in the map the response lies (bottom row of Figure 6.16). This is *global pooling*, and it reduces each channel to one number.

For a single response on a bare map this is true by definition, and no experiment is needed. In a network, however, global pooling summarizes the output of a convolution, and a convolution follows a shift only away from the border (Section 6.1). To see what the border does, we move the single response through all four columns once more and apply a 3×3 kernel of ones with one pixel of zero padding, which copies the response into every output whose window contains it. Then we pool the result globally.

```
pad = lambda Z: F.pad(Z, (1, 1, 1, 1))  # One pixel of zeros on every side
```

```
for j in range(4):
    H = d2l.numpy(d2l.corr2d(pad(single_response(1, j)), d2l.ones((3, 3))))
    print(f'response at (1, {j}): global max {H.max():.2f}, '
          f'global mean {H.mean():.4f}')
```

```
response at (1, 0): global max 1.00, global mean 0.3750
response at (1, 1): global max 1.00, global mean 0.5625
response at (1, 2): global max 1.00, global mean 0.5625
response at (1, 3): global max 1.00, global mean 0.3750
```

The global maximum is 1 wherever the response lies. The global mean, however, is not constant. Nine windows contain a response in column 1 or 2, but only six contain one in the border columns 0 and 3, which is the count of Figure 6.8, and the mean falls from $9/16$ to $6/16$. The conclusion holds beyond this small map: convolutions followed by global pooling are invariant to shifts that keep the content away from the border, and only approximately invariant to the others.

In the networks of Chapter 7 the two tasks of pooling part ways. Halving the resolution is often left to a convolution with stride 2, which learns how to combine the entries of a window instead of fixing the combination to a maximum or a mean, and the network usually ends in *global average pooling* (Lin et al. 2013). LeNet, in the next section, does neither.

6.5.4 Padding and Stride

Here there is little to learn. The pooling layers of the frameworks take a window shape, a stride, and a padding, with the meaning they have for convolutions, and the output shape follows the same rule (Section 6.3): with p rows of padding on each side and stride s, a window of height k on an input of height n gives $\lfloor (n + 2p - k)/s \rfloor + 1$ rows, and likewise for the columns. Only one thing is new, the value that fills the padding. The frameworks expect four-dimensional input, so we put the 4×4 matrix of the numbers 0 to 15 into a tensor with one example and one channel.

```
# (example, channel, height, width)
X = d2l.reshape(d2l.arange(16, dtype=d2l.float32), (1, 1, 4, 4))
X
```

```
tensor([[[[ 0.,  1.,  2.,  3.],
          [ 4.,  5.,  6.,  7.],
          [ 8.,  9., 10., 11.],
          [12., 13., 14., 15.]]]])
```

Pooling summarizes a region, so the usual stride equals the window shape: the windows then tile the input without overlapping. A 3×3 window on the 4×4 input fits once, and the output is the maximum of the top-left 3×3 block, 10. With no parameters, the layer needs no initialization either.

```
pool = nn.MaxPool2d(3)  # The stride defaults to the window shape
pool(X)
```

```
tensor([[[[10.]]]])
```

Overlapping windows need an explicit stride. With padding 1 and stride 2, the output is $\lfloor (4 + 2 - 3)/2 \rfloor + 1 = 2$ entries high and wide.

```
pool = nn.MaxPool2d(3, padding=1, stride=2)
pool(X)
```

```
tensor([[[[ 5.,  7.],
          [13., 15.]]]])
```

Which value fills the padding? A convolution pads with zeros, which add nothing to a weighted sum. For a maximum, the neutral value is $-\infty$: padded positions then never win, and each output is the maximum of the real entries under its window. Here the windows cover the real entries in rows and columns $\{0, 1\}$ and $\{1, 2, 3\}$, so the outputs are 5, 7, 13, and 15. Padding with $-\infty$ also keeps a property that every maximum has: adding a constant to every input entry adds the same constant to every output. We check it on `X - 16`, whose entries all lie below zero.

```
pool(X - 16)
```

```
tensor([[[[-11.,  -9.],
          [ -3.,  -1.]]]])
```

Each output is 16 below the previous one, as it should be. Zero padding would have produced 0 in the three windows that touch the padding, since 0 exceeds every real entry there.

Average pooling is less fortunate: it has no neutral padding value. If the padded positions count as zeros in the mean, they pull the outputs at the border toward 0; the alternative is to average only the real entries under each window.

Average pooling here counts the padded zeros by default (`count_include_pad=True`); `count_include_pad=False` averages only the real entries.

6.5.5 Multiple Channels

A convolutional layer sums over its input channels (Section 6.4): every output channel reads all of them. A pooling layer does nothing of the kind. It pools each input channel separately, so its output has as many channels as its input. To check this, we concatenate `X` and `X + 1` along the channel axis to obtain an input with two channels.

```
X = d2l.concat((X, X + 1), 1)
X
```

```
tensor([[[[ 0.,  1.,  2.,  3.],
          [ 4.,  5.,  6.,  7.],
          [ 8.,  9., 10., 11.],
          [12., 13., 14., 15.]],

         [[ 1.,  2.,  3.,  4.],
          [ 5.,  6.,  7.,  8.],
          [ 9., 10., 11., 12.],
          [13., 14., 15., 16.]]]])
```

The pooling layer with padding 1 and stride 2 from above returns two channels: the first holds the outputs computed earlier, 5, 7, 13, and 15, and the second the same values plus 1.

```
pool(X)
```

```
tensor([[[[ 5.,  7.],
          [13., 15.]],

         [[ 6.,  8.],
          [14., 16.]]]])
```

A pooling layer finds nothing by itself. It has no weights, and whatever it summarizes, a convolution before it had to detect. The parts are now all on the table: a layer that detects, settings for its shape and reach, channels, and a summary of neighborhoods. What remains is to see whether, put together and trained, they classify images better than the MLP did.

6.5.6 Summary

Pooling replaces each window of its input by its maximum or its mean. It has no parameters, it pools each channel separately, and its output shape follows the padding and stride rules of convolutions; max-pooling pads with $-\infty$, so padding never changes a maximum. A 2×2 window with stride 2 halves the resolution, and it absorbs a shift of a response that stays inside its window, but not one that crosses into the next. Max-pooling keeps the strength of the strongest response; average pooling dilutes it over the window. Only a window as large as the feature map, global pooling, removes the position of a response altogether, and after a padded convolution it does so only up to the border.

6.5.7 Exercises

1. code **Average pooling as convolution.** Express $k_\mathrm{h} \times k_\mathrm{w}$ average pooling with stride $(s_\mathrm{h}, s_\mathrm{w})$ on an input with c channels as a convolution with c output channels. Give the kernel $\mathbf{K}_{o,c}$ for $o = c$ and for $o \neq c$, so that each channel is pooled separately, as

pooling layers do. On the input of Figure 6.15, verify that your convolution reproduces `pool2d(X, (2, 2), 'avg')`.

2. **Receptive field and cost of downsampling.** Consider 3×3 convolutions with stride 1 and padding 1, with 64 input and 64 output channels, on a 224×224 input.
   1. How many such layers does a stack need before the receptive field of a unit at its top covers the whole image?
   2. Now insert 2×2 max-pooling with stride 2 after every second convolutional layer. How many convolutional layers are needed now?
   3. Count the multiply-adds of the two stacks, and the comparisons that the pooling layers perform. Which part of the cost does pooling remove, and what does it cost itself?

3. **Pooling alternatives.**
   1. Do we need a separate minimum-pooling layer, or can it be expressed through an operation already in this section?
   2. *Softmax pooling* with temperature $\tau > 0$ replaces the window entries $x_1, \dots, x_m$ by their softmax-weighted average

$$\sum_{i=1}^{m} x_i \frac{\exp(x_i/\tau)}{\sum_{j=1}^{m} \exp(x_j/\tau)}. \tag{6.18}$$

   Find its limits as $\tau \to 0$ and as $\tau \to \infty$. Give one cost that softmax pooling adds relative to max-pooling.

4. **Aliasing and blur-pool.** Naive stride-2 downsampling keeps every second entry of its input. Measure frequency in cycles per input sample, so that the one-dimensional signals $(1, 0, 1, 0, 1, 0)$ and $(0, 1, 0, 1, 0, 1)$ both have frequency $\frac{1}{2}$.
   1. Apply naive stride-2 downsampling to both signals, which differ by a shift of one position, and compare the two outputs. Why is this behavior called *aliasing*?
   2. Which input frequencies can a stride-2 subsampler represent faithfully?
   3. Blur-pool (Zhang 2019) counteracts aliasing by applying a small averaging filter before subsampling. Work out what blur-pool with a two-tap box filter $(\frac{1}{2}, \frac{1}{2})$ computes on the two signals above. Which pooling operation from this section does it coincide with?

5. code **Backward pass of max-pooling.** Derive the gradient of two-dimensional max-pooling with respect to its input, for a $k_\mathrm{h} \times k_\mathrm{w}$ window and stride equal to the window size, and implement it without automatic differentiation. Verify against the gradient that automatic differentiation computes through the built-in max-pooling operation of this section on a small random tensor, to a relative error below 10^{-4}.

   Adapted from Stanford CS231n, Assignment 2, ConvolutionalNetworks, naive max-pooling backward pass.

6. **Max-pooling via ReLU.** ⊠ A convolution is linear, so it cannot compute a maximum by itself.
   1. Express $\max(a, b)$ using only affine operations and $\mathrm{ReLU}(x) = \max(0, x)$.
   2. Use this to implement max-pooling as a stack of convolutions and ReLU layers.

3. Count the channels per layer and the ReLU layers that your construction uses for a 2×2 window and for a 3×3 window. How does the number of ReLU layers grow with the number of entries in the window?

6.6 LeNet

The preceding sections supplied the parts: the convolutional layer (Section 6.2), padding and stride (Section 6.3), channels (Section 6.4), and pooling (Section 6.5). None of them has classified anything yet. We now assemble them into a complete classifier and ask the obvious question: is it any better than the MLP?

The number to beat is that of the MLP of Section 4.2, about 87% validation accuracy on Fashion-MNIST after 30 epochs. Recall that the MLP reads an image as 784 numbers in no particular order (Section 6.1), whereas convolutional layers keep the spatial layout and use the same kernel at every position. To keep the comparison fair, we train the convolutional classifier with the same loss, optimizer, and number of epochs as the MLP and compare their validation accuracy. One number will not settle the matter, however. We therefore also look at what the first layer has learned, and we repeat the comparison on images whose pixels have been shuffled.

The network we build is *LeNet*, after Yann LeCun, who developed it with his colleagues at AT&T Bell Labs to recognize handwritten digits (LeCun, Léon Bottou, et al. 1998). It was the result of a decade of work that began with an early convolutional network for images trained end to end by backpropagation (LeCun et al. 1989). LeNet-5, the version of 1998, reached an error rate below 1% on handwritten digits, on par with the support vector machines of the time, and variants of it read the handwritten amounts on bank checks in commercial systems. It showed that a network trained from examples could replace a hand-engineered recognition pipeline.

6.6.1 The LeNet Architecture

LeNet consists of two parts, shown in Figure 6.17: a convolutional encoder with two convolutional layers, and a dense block with three fully connected layers.

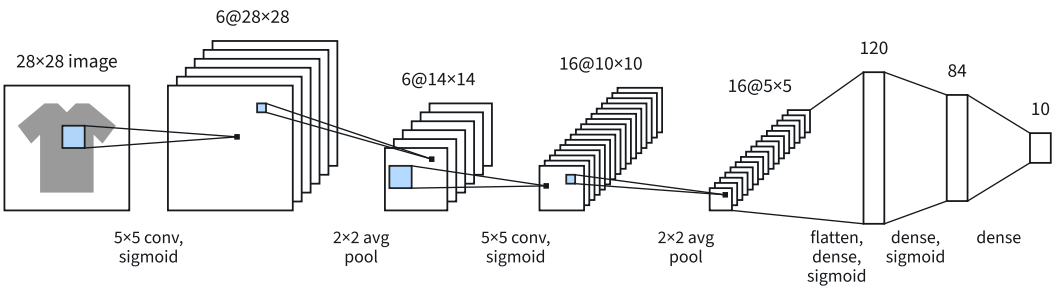


Figure 6.17: Data flow in LeNet on one 28×28 Fashion-MNIST image, with the feature maps drawn to scale. Each shaded window produces the dark unit it points to in the next layer. The three fully connected layers have 120, 84, and 10 outputs.

Each convolutional block consists of a convolutional layer with a 5×5 kernel, a sigmoid activation, and 2×2 average pooling with stride 2. The first convolutional layer has 6 output

channels and the second has 16.

The encoder's output has shape (batch size, channels, height, width).

The dense block needs one vector per example, so we first flatten the encoder's output into a matrix with one row per example. Its three fully connected layers have 120, 84, and 10 outputs. The last layer has one output per class, as in softmax regression.

This is the common teaching version of LeNet-5, not an exact reproduction. The network of 1998 had the same alternating convolutions and pooling and the same widths of 6, 16, 120, and 84. It differed in details that need not concern us: a scaled tanh as activation, pooling with a trainable weight and bias, partial connections between the two convolutional layers, and radial-basis output units. One difference does leave a trace in our code. The original took 32×32 images, so our first convolutional layer pads its 28×28 input by two pixels on each side, which gives it the same extent and the same 28×28 output.

In a framework, LeNet is a `Sequential` container of these layers. We use Xavier initialization (Section 4.4.2.1) for the weights and zero biases.

Lazy layers have no weights until the first minibatch fixes their shapes (Section 5.1.4), so the `Trainer` initializes them after that minibatch, by calling the model's `weight_init` method on every layer. Ours applies `init_cnn`, which draws Xavier-uniform weights, and sets the biases to zero.

```
def init_cnn(module):
    """Initialize weights for CNNs."""
    if type(module) == nn.Linear or type(module) == nn.Conv2d:
        nn.init.xavier_uniform_(module.weight)
```

```
class LeNet(d2l.Classifier):
    """The layout of LeNet-5, with sigmoid activations and average pooling."""
    def __init__(self, lr=0.1, num_classes=10):
        super().__init__()
        self.save_hyperparameters()
        self.net = nn.Sequential(
            nn.LazyConv2d(6, kernel_size=5, padding=2), nn.Sigmoid(),
            nn.AvgPool2d(kernel_size=2, stride=2),
            nn.LazyConv2d(16, kernel_size=5), nn.Sigmoid(),
            nn.AvgPool2d(kernel_size=2, stride=2),
            nn.Flatten(),
            nn.LazyLinear(120), nn.Sigmoid(),
            nn.LazyLinear(84), nn.Sigmoid(),
            nn.LazyLinear(num_classes))

    def weight_init(self, module):
        # The Trainer calls this after the first minibatch fixed the shapes
        init_cnn(module)  # Xavier-uniform weights
        if type(module) in (nn.Linear, nn.Conv2d) and module.bias is not None:
            nn.init.zeros_(module.bias)
```

6.6.2 Layer Shapes

A wrong padding or a missing flattening step changes the shape of every layer that follows it, so we check the shapes before training. Figure 6.18 lists the layers in compressed notation, with the shape of the tensor that passes between them. Does the code build this network? To find out, we pass a single-channel 28×28 image through it and print the output shape after each layer. The method `layer_summary` does this for any classifier whose layers sit in a sequential container `net`; the next chapter uses it for larger networks.

```
@d2l.add_to_class(d2l.Classifier)
def layer_summary(self, X_shape):
    X = d2l.randn(*X_shape)
    for layer in self.net:
        X = layer(X)
        name = getattr(layer, '__name__', type(layer).__name__)
        print(f'{name:<22} {tuple(X.shape)}')

model = LeNet()
model.layer_summary((1, 1, 28, 28))
```

```
Conv2d                 (1, 6, 28, 28)
Sigmoid                (1, 6, 28, 28)
AvgPool2d              (1, 6, 14, 14)
Conv2d                 (1, 16, 10, 10)
Sigmoid                (1, 16, 10, 10)
AvgPool2d              (1, 16, 5, 5)
Flatten                (1, 400)
Linear                 (1, 120)
Sigmoid                (1, 120)
Linear                 (1, 84)
Sigmoid                (1, 84)
Linear                 (1, 10)
```

It does, and the shapes follow the rules of Section 6.3. A 5×5 kernel without padding shrinks the height and the width by 4. The first convolutional layer, with its padding of 2, keeps 28×28; the second has none and turns 14×14 into 10×10. Each pooling layer halves the height and the width, from 28 to 14 and from 10 to 5. Meanwhile the number of channels grows from 1 to 6 to 16, so the encoder trades spatial resolution for channels. The flattened output has $16 \times 5 \times 5 = 400$ entries, which the dense block reduces to one output per class.

6.6.3 Parameters and Multiply-Adds

With the shapes in hand we can say what every layer costs. Recall from Section 6.4.2 that a convolutional layer with c_i input channels, c_o output channels, and a $k \times k$ kernel has $c_\mathrm{i} c_\mathrm{o} k^2$ weights and c_o biases, and that it applies each weight once at every output position. A fully connected layer uses each of its weights only once per example. Table 6.1 applies these counts to LeNet.

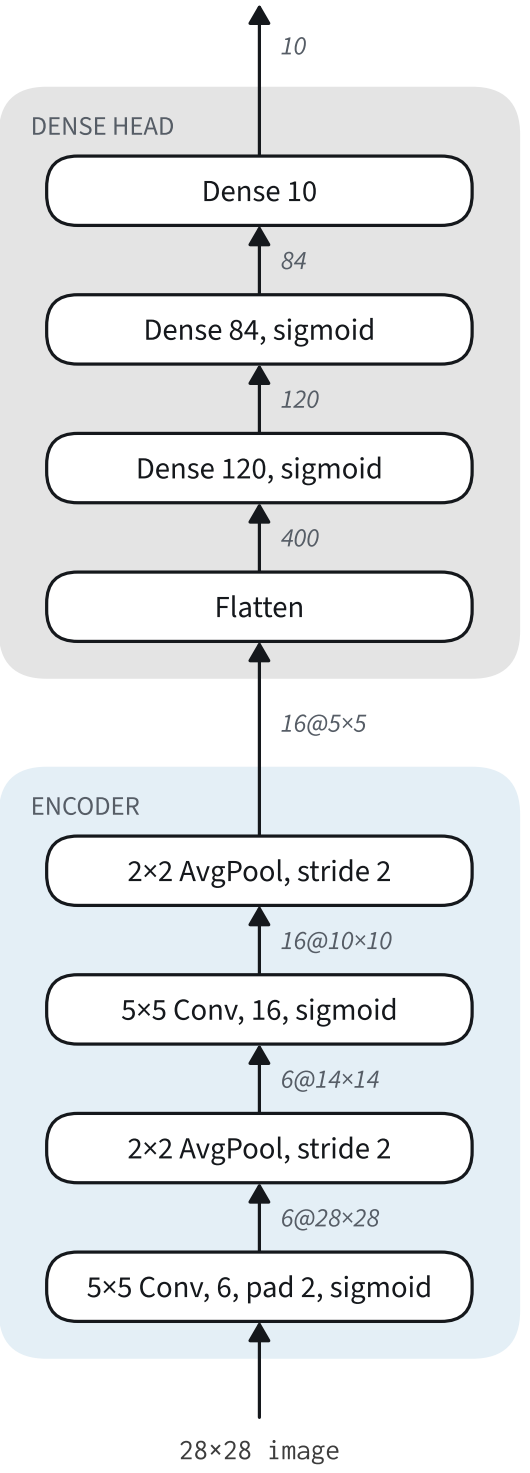


Figure 6.18: Compressed notation for LeNet. Beside each arrow is the shape of the tensor it carries, as channels@height×width.

Table 6.1: Parameters and multiply-adds per example of the layers of LeNet. Pooling and activation layers have no parameters.

layer	output	parameters	multiply-adds
5×5 convolution, 1 to 6 channels	6@28×28	$150 + 6 = 156$	$150 \times 784 = 117{,}600$
5×5 convolution, 6 to 16 channels	16@10×10	$2{,}400 + 16 = 2{,}416$	$2{,}400 \times 100 = 240{,}000$
fully connected, 400 to 120	120	$48{,}000 + 120 = 48{,}120$	$48{,}000$
fully connected, 120 to 84	84	$10{,}080 + 84 = 10{,}164$	$10{,}080$
fully connected, 84 to 10	10	$840 + 10 = 850$	840
total		$61{,}706$	$416{,}520$

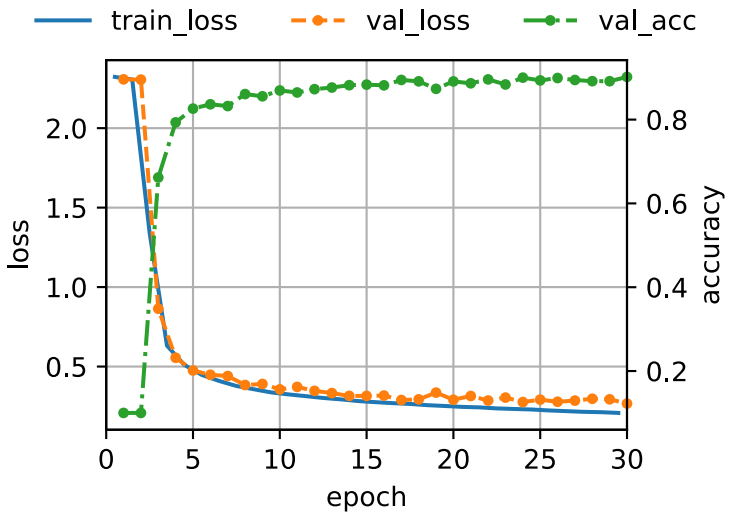
The two kinds of layers divide the work in opposite ways. The convolutional layers hold 4% of the parameters and perform 86% of the multiply-adds, because each of their weights is used at every position of a feature map. The first fully connected layer holds 78% of the parameters and performs 12% of the multiply-adds. Compared with the MLP, which has 203,530 parameters and performs about 203,000 multiply-adds per image, LeNet has fewer than a third of the parameters and performs about twice the arithmetic.

6.6.4 Training

As for the MLP, we minimize the cross-entropy by minibatch stochastic gradient descent for 30 epochs. The minibatches hold 128 images, half the MLP's 256. Unfortunately, the learning rate cannot stay where it was. The trouble is the sigmoid, which passes back at most $1/4$ of the gradient (Section 4.1.4). The gradient that reaches the weights of the first convolutional layer has passed backward through four sigmoids, one after each of the four hidden layers. At the MLP's learning rate of 0.1, LeNet learns slowly: its validation accuracy stays at chance for the first two to four epochs and reaches only about 73% after 10. We use a learning rate of 2.0 instead. The factor of twenty is empirical: the argument says only that the rate must grow.

As in Section 5.7.4, the `Trainer` runs on the GPU if there is one.

```
trainer = d2l.Trainer(max_epochs=30, num_gpus=1)
data = d2l.FashionMNIST(batch_size=128)
model = LeNet(lr=2.0)
trainer.fit(model, data)
```



After 30 epochs LeNet reaches a validation accuracy of about 88–90%, against about 87% for the MLP.

The start of the curve deserves a second look. In some runs the accuracy rises at once. In others it sits at chance, 10%, for a few epochs, rarely more, with the loss stuck near $\ln 10 \approx 2.3$, and then takes off. Nothing distinguishes these runs but their random initial weights. While the loss stays at $\ln 10$ the network gives every image the same ten probabilities: after four sigmoids, the part of its output that depends on the input is still too small to matter. An exercise measures this, and the plateau will return below.

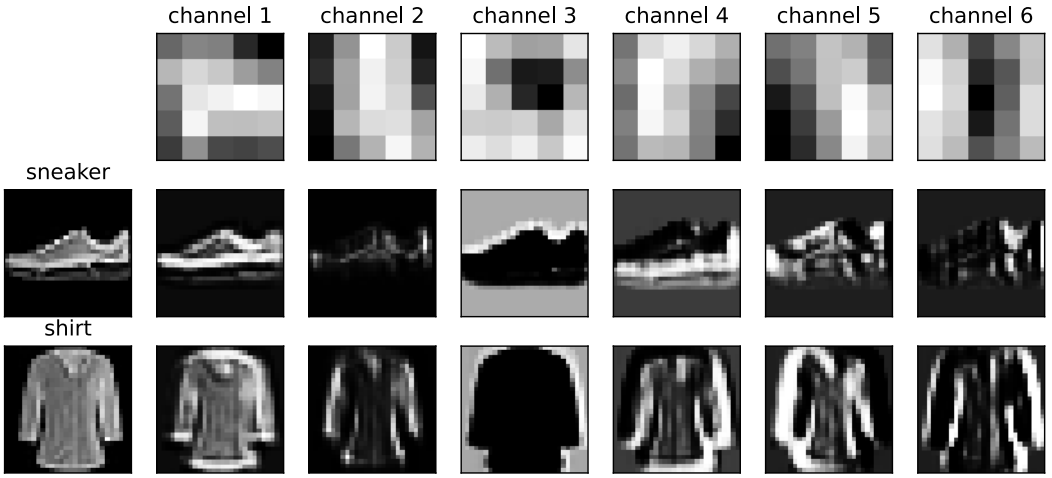
6.6.4.1 The Learned Features

For softmax regression the weights were ten templates, one blurred garment per class (Section 3.3), and the hidden units of the MLP weighed regions of the image (Section 4.2). Both could be drawn as 28×28 pictures, because every weight belonged to one pixel. A kernel of LeNet belongs to no pixel in particular. It is a 5×5 array of weights, too small to tell the eye much, so we look at what it does as well as at what it is. We take a sneaker and a shirt from the validation set and draw the six kernels of the first convolutional layer in the top row, and below each kernel the feature map that it produces for the two garments, after the sigmoid. Every image has its own gray scale, from its smallest value (black) to its largest (white).

```
X, y = trainer.prepare_batch(next(iter(data.val_dataloader())))
labels = d2l.numpy(y)
X = d2l.stack([X[int(np.argmax(labels == c))] for c in (7, 6)], 0)
```

```
maps = model.net[1](model.net[0](X))  # The first convolution and its sigmoid
images, maps = d2l.numpy(X[:, 0]), d2l.numpy(maps)
kernels = d2l.numpy(model.net[0].weight[:, 0])  # Six 5x5 kernels
```

```
names = data.text_labels([7, 6])
axes = d2l.show_images(
    [kernels[0], *kernels, images[0], *maps[0], images[1], *maps[1]], 3, 7,
    titles=['', *[f'channel {c}' for c in range(1, 7)],
            names[0], *[''] * 6, names[1], *[''] * 6], cmap='gray')
axes[0].set_visible(False)  # No kernel above the input images
```



Two kinds of channels appear, and they appear in every run. Some reproduce the silhouette of the garment, filled in, either bright on a dark background or inverted. Such a channel reports, roughly, whether its neighborhood is bright or dark. The others light up only along the outline, most of them on its left or on its right side. They respond to vertical edges, and several of their kernels show how: a column of bright weights beside a column of dark ones, a wider relative of the kernel $(1, -1)$ of Section 6.2.2.2. There we had to hand gradient descent the edge map before it found the detector. Here nobody asked for edges. The network was given garments and their labels, and found that edges help to tell them apart. Which channel does what changes from run to run.

6.6.4.2 Shuffled Pixels

How much should we read into LeNet's gain of one to three points over the MLP? Not too much. The MLP's own accuracy moves by about a point from one epoch to the next (Section 4.2), and LeNet differs from the MLP in its depth, its activation function, and its learning rate as well as in its convolutions. A sharper test removes the one thing that only the convolutions know about, namely which pixels are neighbors. This is the shuffling that the chapter introduction imagined: we fix one random order of the 784 pixel positions and rearrange every image, for training and validation alike, into that order.

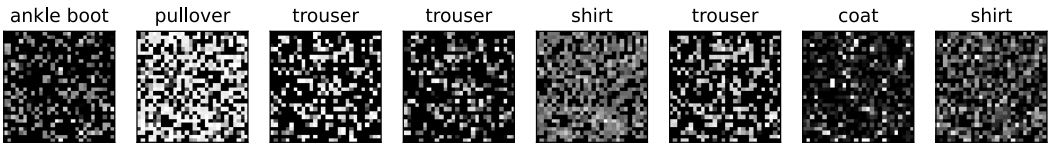
```
perm = np.random.permutation(28 * 28)  # One fixed order for every image

def shuffle_pixels(images):
    """Rearrange the 784 pixels of every image into the order perm."""
    return images.reshape(len(images), -1)[:, perm].reshape(images.shape)
```

```
data_shuffled = d2l.FashionMNIST(batch_size=128)
for split in (data_shuffled.train, data_shuffled.val):
    split.data = shuffle_pixels(split.data)  # The stored 8-bit images
```

To us the shuffled images are noise.

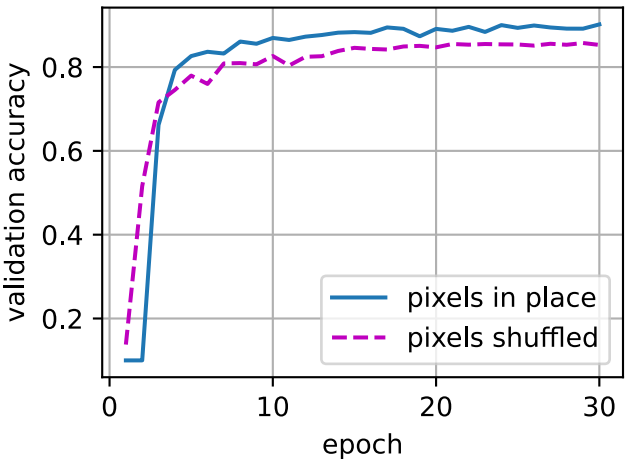
```
data_shuffled.visualize(next(iter(data_shuffled.val_dataloader())))
```



The MLP cannot tell the difference, and we need not train it again to know this. Shuffling the pixels only reorders the inputs of its first layer, so training on shuffled images is training on the original ones with the columns of $\mathbf{W}^{(1)}$ reordered. Since every weight starts from the same distribution, it is the same random experiment, and the MLP would again reach about 87%. LeNet, however, now applies its kernels to 5×5 windows of pixels that were never neighbors. We train it with the same settings as before and plot both validation curves.

```
curves = {'pixels in place': trainer.history['val_acc']}
trainer.board.display = False
trainer.fit(LeNet(lr=2.0), data_shuffled)
curves['pixels shuffled'] = trainer.history['val_acc']
d2l.plot(list(range(1, trainer.max_epochs + 1)), list(curves.values()),
         'epoch', 'validation accuracy', legend=list(curves))
for name, acc in curves.items():  # A single epoch can land a point low
    print(f'{name}: {sum(acc[-5:]) / 5:.3f} averaged over the last 5 epochs')
```

```
pixels in place: 0.896 averaged over the last 5 epochs
pixels shuffled: 0.854 averaged over the last 5 epochs
```



On shuffled pixels LeNet ends several points lower than on the ordered images, at about 85

or 86% in a typical run, which is at or a little below the MLP. Its advantage is gone, and so is the doubt about where the advantage came from: it was the advantage of knowing which pixels are adjacent.

The plateau at chance is back as well, and it is as unpredictable as before. On ordered pixels it lasted at most a few epochs. On shuffled pixels it usually does the same, but once in a while it takes up most of the thirty epochs, and such a run ends lower still. A reader who repeats the experiment may well meet one. The plateau is a weakness of this network of sigmoids and does not affect the comparison: in none of our runs, however short its plateau, did LeNet on shuffled pixels end above the MLP.

One more observation ties LeNet back to the derivation. The network of Section 6.1 was to be equivariant in its layers and invariant in its answer, and LeNet is neither. Its convolutions follow a shift only away from the border, and its pooling layers absorb some one-pixel shifts and pass on others (Section 6.5). Its head then flattens the 5×5 maps and gives every position weights of its own, so nothing makes the answer invariant. What tolerance to shifts LeNet has is partial, and Fashion-MNIST, whose garments are all centered, never asks for more.

The two runs together show that the accuracy of the MLP can be had, and slightly improved on, with fewer than a third of its parameters, and that the improvement comes from the convolutions' knowledge of the grid. A kernel learns a local pattern once and detects it at every position, whereas the MLP must learn it anew wherever it occurs. On 28×28 images little is at stake. On the megapixel photographs of Section 6.1, where a single fully connected layer needs a billion weights or more, the same idea decides whether the network can be built at all. Whether such a network can also be trained is another matter, and the four sigmoids above give a first hint of the difficulty.

6.6.5 Summary

LeNet combines a convolutional encoder, in which the resolution shrinks while the number of channels grows, with a small fully connected head. Its convolutional layers hold few parameters and do most of the arithmetic; the first fully connected layer holds most of the parameters. Trained like the MLP of Section 4.2, it reaches about 88–90% validation accuracy on Fashion-MNIST, against about 87% for the MLP, with fewer than a third of its parameters. Its first layer learns silhouettes and vertical edges from the labels alone. When the pixels are shuffled, the advantage over the MLP disappears, so it came from the convolutions' knowledge of the grid. The four sigmoids, each passing back at most $1/4$ of the gradient, call for a much larger learning rate, 2.0 against 0.1, and they make the start of training erratic.

The encoder followed by a classifier head remains the organization of most convolutional networks, but nearly every component of LeNet has since been replaced: the sigmoid by the ReLU, average pooling by max-pooling or strided convolutions, and the dense head by global average pooling and a single linear layer, with normalization layers added throughout. The next chapter makes these changes one at a time, starting with AlexNet (Section 7.1), and asks what each one contributes as networks grow much deeper.

6.6.6 Exercises

1. code **Modern components.** Retrain `LeNet` with the changes below, using the training settings of this section, and compare the validation accuracy of each variant after 30

epochs with that of `LeNet`.

1. Replace average pooling with max-pooling.
2. Replace the sigmoid activations with ReLU. If the loss diverges or the accuracy stays at chance, lower the learning rate until training succeeds, and report the learning rate you used.
3. Make both changes together, and call the result the modernized LeNet. Which of the two changes accounts for most of the difference in accuracy?
4. ⊠ With the weights the model has at the start of training, measure for `LeNet` and for the modernized LeNet the norm of the gradient of the loss with respect to the first convolutional kernel, and the standard deviation of the logits across a minibatch. The derivation of Xavier initialization in Section 4.4.2.1 assumes inputs with zero mean and no nonlinearity. Use the mean of the sigmoid's output and its slope $\sigma'(0) = 1/4$ to explain your measurements, the learning rate that `LeNet` needs, and the epochs at chance at the start of its training.

2. code **Original MNIST.** Write an `MNIST` data class by adapting `d2l.FashionMNIST`, and train both `LeNet` and the MLP of Section 4.2 on it, with the settings that each uses for Fashion-MNIST. LeCun, Léon Bottou, et al. (1998) report a test error rate below 1% for LeNet-5 on MNIST.

    1. Compare the validation error rate of `LeNet` with this figure. The two are measured on the same images: as for Fashion-MNIST, our validation set is the dataset's official test set.
    2. For each dataset, compute the difference and the ratio of the validation error rates of the two networks. Section 3.2 states that MNIST makes differences among model classes less visible. Does each of the two measures support this statement?

3. code **Deeper feature maps.** The text drew the feature maps of the first convolutional layer.

    1. Display the 16 feature maps of the second convolutional layer of the trained `LeNet`, after its sigmoid, for validation images of two classes that are easily confused, for example pullovers and coats. Can you still say what a channel responds to? Which channels differ most between the two classes?
    2. Train the modernized LeNet of problem 1 and measure the maximum activation of each of its two convolutional layers, taken after the ReLU, on Fashion-MNIST validation images, on out-of-distribution photos such as a cat or a car, and on uniform random noise. Convert each photo to a 28×28 grayscale image, and keep every input in $[0, 1]$, the pixel range of `d2l.FashionMNIST`. Do the maxima for photos and noise fall inside or clearly outside the range observed on the validation images?

4. code **Overfitting sanity check.** Before trusting a full training run, check that a network can memorize a small dataset. Fix a subset of 50 Fashion-MNIST training images and train on it with all 50 images in one minibatch, for at most 500 gradient steps.

    1. For both `LeNet` and the modernized LeNet of problem 1, record the number of steps until training accuracy exceeds 99%. If a network does not get there, diagnose which architectural or optimization choice is responsible.
    2. Name one bug that this check would catch and one that it would miss, and explain why the check cannot detect the second.

Adapted from Stanford CS231n, Assignment 2, Convolutional Networks notebook.

5. code **Shifted garments.** The text argued that `LeNet` is only partly tolerant to shifts.
   1. Shift every validation image by one, two, and four pixels to the right, filling the vacated columns with zeros, and measure the validation accuracy of the trained `LeNet` on each shifted set. Which shifts does the network tolerate? Relate what you find to the two pooling layers and to the dense head.
   2. Explain why training `LeNet` on translated copies of its training images can improve the accuracy on shifted validation images even though its convolutional layers share weights across all locations.

   Adapted from Michael Nielsen, Neural Networks and Deep Learning, Chapter 6.

6. code **The dense head.** ☒ Table 6.1 shows that the first fully connected layer holds 78% of LeNet's parameters.
   1. Replace the entire dense head, from the flatten through the output layer, by global average pooling followed by a single linear layer from 16 to 10 units. How many parameters does the network now have? Before training, predict whether the validation accuracy changes by more than one percentage point relative to `LeNet`; then retrain and compare.
   2. Instead connect the flattened 400-dimensional convolutional output directly to a ten-way linear output layer. How many parameters does this network have? Retrain, and compare the accuracy of the two reduced heads with each other and with your prediction. What does the comparison say about where LeNet's capacity is used?

   Adapted from Michael Nielsen, Neural Networks and Deep Learning, Chapter 6.

Chapter 7

Modern Convnets

The LeNet of Section 6.6 learns its convolution kernels from examples, and it classifies Fashion-MNIST slightly better than the MLP with fewer than a third of its parameters. A natural next step is to make it larger, with more layers and more channels. For a long time this did not work. Deeper stacks of sigmoid layers trained worse than shallow ones, since every sigmoid shrinks the gradient that passes back through it, and every new layer brought its own kernel size, width, and initialization to choose by hand. What does it take to make a convolutional network much deeper, and better?

The network that won the ImageNet competition in 2012 had 8 layers, and the winner three years later had more than 150. ImageNet is a benchmark of more than a million photographs in a thousand categories, and the 2012 winner, a convolutional network trained end to end on raw pixels, beat every system built on hand-designed features. Between the two networks lies a series of ideas that removed one obstacle at a time: repeated blocks, layers that keep the scale of the signals under control, and connections that let each block add a correction to its input instead of replacing it. The last two reappear in every transformer.

Later gains came as much from how networks are trained as from new layers. This is awkward for anyone who compares two published accuracies, because the better number may belong to the better training recipe and not to the better network.

The first four sections follow the growth in depth: the network of 2012 (Section 7.1), networks built from repeated blocks instead of layer by layer (Section 7.2), the normalization of activations (Section 7.3), and the residual connection (Section 7.4), where we train plain and residual networks side by side to see that depth alone was not enough. Section 7.5 goes the other way and makes a network cheap enough for a phone, then asks what the saving is worth against a plain network of equal cost. From there on the chapter is about evidence as much as about layers. We ask how much of a reported accuracy comes from the training procedure (Section 7.6), modernize a ResNet a few changes at a time to find out which of them matter (Section 7.7), and sample two small populations of networks to test a design rule (Section 7.8).

Every network in the chapter is trained on Fashion-MNIST, under one learning-rate schedule that Section 7.1 defines. A benchmark this small shows that a network trains and what it costs. The evidence that an idea matters at scale comes from the ImageNet results, which we quote from the papers. Vision transformers (Dosovitskiy et al. 2021) follow in Chapter 11. With equal compute, convnets match them on image classification (S. L. Smith et al. 2023),

and they remain the default where data are limited, latency matters, or every pixel needs a prediction.

Resources and Further Reading

The references below cover the architectures of this chapter and current practice: the landmark networks from AlexNet to ConvNeXt, the training recipes that confound comparisons between them, and efficient networks for deployment. All are freely accessible online.

Books

- Deep Learning for Coders with fastai and PyTorch, by Howard and Gugger. Free notebooks that build ResNets from scratch and apply the augmentation, schedules, and mixed precision of Section 7.6 in working code.
- Understanding Deep Learning, by Simon J. D. Prince. Chapter 11 analyzes why residual connections ease optimization, with loss-surface and gradient-propagation arguments that complement Section 7.4.

Courses and video lectures

- Michigan EECS 498-007, lecture on CNN architectures, by Justin Johnson. Covers the networks of the first four sections, from AlexNet to ResNet, with the same accounting of parameters and arithmetic.
- MIT 6.5940: TinyML and Efficient Deep Learning, by Song Han. Depthwise separability, quantization, pruning, and deployment on small devices: the systems side of Section 7.5.
- Hugging Face Computer Vision Course. Current practice with pretrained backbones, ConvNeXt among them, and transfer learning.

Tutorials, notes, and surveys

- timm (pytorch-image-models), by Ross Wightman and contributors. Implementations of the architectures in this chapter with trained weights, and results tables that compare them under one evaluation.
- A Recipe for Training Neural Networks, by Andrej Karpathy. The discipline of debugging first, which Section 7.6 assumes.
- Zoom In: An Introduction to Circuits, by Olah et al. (2020), *Distill*. Opens trained vision models to inspect the features they learn; a natural next step after the first-layer kernels of Section 7.1.

Foundational papers

A selection of the primary sources for this chapter, all free on arXiv or the proceedings site and worth reading in the original. The papers on VGG, NiN, GoogLeNet, and RegNet are cited in their sections.

- ImageNet Classification with Deep Convolutional Neural Networks, by Krizhevsky, Sutskever, and Hinton (2012), *NeurIPS*. AlexNet (Section 7.1).
- Batch Normalization, by Ioffe and Szegedy (2015). Read it together with the later critique of its explanation in Section 7.3.
- Deep Residual Learning for Image Recognition, by He, Zhang, Ren, and Sun (2015). ResNet (Section 7.4), the most cited paper in deep learning.
- Bag of Tricks for Image Classification with CNNs, by He et al. (2019). The first systematic demonstration that details of the recipe rival changes of architecture (Section 7.6).

- ResNet Strikes Back, by Wightman, Touvron, and Jégou (2021). The controlled study of recipes and architecture behind Section 7.6.
- MobileOne: An Improved One Millisecond Mobile Backbone, by Vasu et al. (2023), *CVPR*. Re-parameterized networks designed against measured phone latency; continues Section 7.5 toward deployment.
- A ConvNet for the 2020s, by Liu et al. (2022). ConvNeXt (Section 7.7), a cumulative ablation worth studying for its method as much as for its result.
- ConvNets Match Vision Transformers at Scale, by Smith et al. (2023). With matched pretraining compute, convnets and vision transformers scale alike; read it with Chapter 11.

7.1 AlexNet and Learned Image Representations

The LeNet of Section 6.6 learns everything it computes from examples, its convolution kernels as well as its classifier. LeCun and his colleagues trained the first such network in 1989, yet for the next two decades most vision systems worked differently. They separated representation from prediction: hand-designed operators such as SIFT (Lowe 2004), SURF (Bay et al. 2006), HOG (Dalal and Triggs 2005), and bags of visual words (Sivic and Zisserman 2003) turned pixels into feature vectors, and a linear or kernel method learned a classifier on top. The training loss could improve the classifier but never the features, so progress depended on people inventing better features. We call a representation learned when the loss decides what the features compute as well. Why did learned features take so long to beat engineered ones, and what changed when they did?

The change came at the ImageNet challenge of 2012, which asked a system to sort photographs into 1000 categories and scored it by its *top-5 error*, the fraction of validation images whose correct label is not among the system's five best guesses. AlexNet (Krizhevsky et al. 2012) replaced the two-stage pipeline with one deep convolutional network trained end to end, and reached a top-5 error of 15.3%, against 26.2% for the best entry built on engineered features. The year before, the winning error had fallen by about two points. This time it fell by more than ten.

AlexNet's architecture is the template of LeNet scaled up: convolutional stages that shrink the resolution and grow the number of channels, followed by fully connected layers. The template was more than twenty years old, which only sharpens the question. Three things made the scale-up work in 2012 when it had not before: a labeled dataset large enough for tens of millions of parameters, GPUs fast enough to train them in days, and techniques (ReLU activations, dropout, data augmentation) that kept a network of this size trainable and kept it from overfitting.

This section explains why each of the three mattered, builds AlexNet, counts where its parameters and its arithmetic sit, and trains it on Fashion-MNIST.

7.1.1 Learned Features, Data, and Hardware

Learning the features was not a new idea in 2012. Besides LeNet, sparse coding had learned image filters without any labels (Olshausen and Field 1996). What was new is that the learned features won on a large benchmark, and AlexNet's first layer shows what the network chose to compute. Figure 7.1 displays its 96 learned kernels of 11×11 pixels: oriented edges at several frequencies and patches of color, close to the oriented-gradient operators that SIFT and HOG compute by design. Nobody told the network to look for edges. It found

these operators because they reduce the classification loss, and it found the operators of its higher layers the same way, where no designer had a good guess to offer.

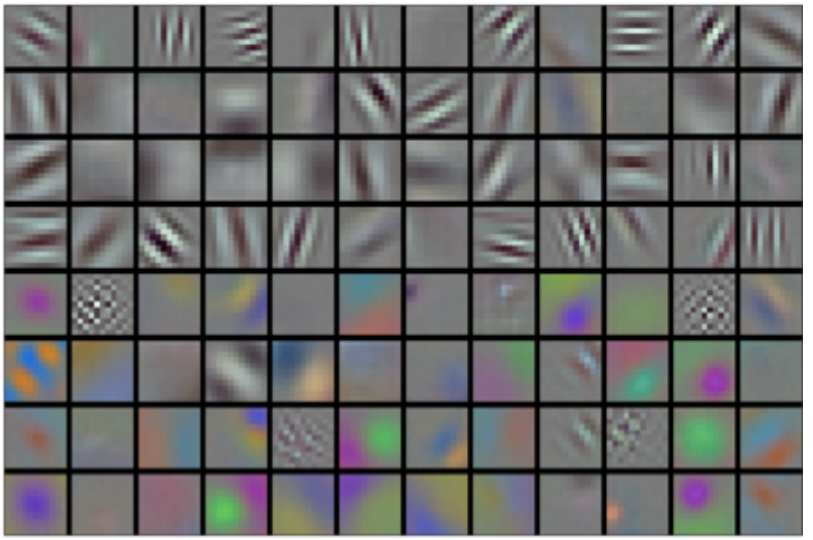


Figure 7.1: The 96 kernels of the first convolutional layer of AlexNet, learned on ImageNet. Reproduction courtesy of Krizhevsky et al. (2012).

7.1.1.1 Data

A model with tens of millions of parameters, fitted to the ten thousand images of a benchmark such as PASCAL VOC, simply memorizes them. The ImageNet dataset (Deng et al. 2009) changed the scale: its 2012 challenge (Russakovsky et al. 2015) supplied about 1.2 million labeled training images in 1000 categories. Even that leaves some fifty parameters for every training image, so AlexNet's authors stretched the data further. They trained on random 224×224 crops of images rescaled to 256×256, and on their mirror images. A crop can start at any of about 32×32 offsets and can be mirrored or not, so each image yields about $32 \cdot 32 \cdot 2 = 2{,}048$ different inputs. These are overlapping views of one photograph, of course, and even with them the network overfit without dropout (Krizhevsky et al. 2012).

7.1.1.2 Hardware

Training a deep network repeats the same small computations at enormous scale. A convolution applies one kernel at every output position and every channel, and each of these products is independent of the others. A *graphics processing unit* (GPU) is built for such work: instead of a few fast, complicated cores it has hundreds or thousands of simple ones and a much wider memory bus. Each of the two NVIDIA GTX 580s on which AlexNet was trained performs about 1.6 trillion floating-point operations per second (TFLOPS) in single precision and reads 192 GB of memory per second; a quad-core desktop CPU of 2011 peaked at about 0.2 TFLOPS and 20 GB/s.

A short calculation shows why this mattered. A network of AlexNet's size needs about a billion multiply-adds to classify one 224×224 color image. A training step costs about three times as many: one pass forward and, on the way back, two products of the same size in each layer, one for the gradient of its weights and one for the gradient handed to the layer

below. With two floating-point operations per multiply-add, the 90 passes over 1.2 million images used in the paper take about $10^9 \cdot 3 \cdot 2 \cdot 1.2 \times 10^6 \cdot 90 \approx 6.5 \times 10^{17}$ operations. The two GPUs need two to three days for that at their peak, and the actual run took five to six days. A CPU at its peak would have needed more than a month, for every attempt. Alex Krizhevsky wrote the GPU code for the convolutions himself (cuda-convnet); today the same network takes a dozen lines, as below.

7.1.2 The AlexNet Architecture

Figure 7.2 places AlexNet beside the LeNet of Section 6.6, both in the versions that this book trains on Fashion-MNIST. Both consist of convolutional stages followed by a fully connected head. AlexNet changes the size of every part and three of its components: the activation function, the pooling, and the regularizer.

7.1.2.1 Depth and Width

AlexNet has eight layers with weights: five convolutional layers and three fully connected layers, against LeNet's two and three. Its images are eight times wider and taller than Fashion-MNIST's, so the first layer uses an 11×11 kernel with stride 4 to cover a large patch of the image at once and to reduce the resolution early. The second layer uses 5×5 kernels and the last three 3×3 kernels. Max-pooling with 3×3 windows and stride 2 follows the first, second, and fifth convolutional layers. It replaces LeNet's average pooling, because a maximum keeps a strong response that an average would dilute (Section 6.5), and its windows overlap. The widths grow accordingly: 96 to 384 channels where LeNet has 6 and 16, and 4096 hidden units where LeNet has 120 and 84.

Our version leaves out two features of the paper's network, and neither is missed. The paper split every layer between two GPUs, because one GPU with 3 GB of memory could not hold the model; a single GPU now can. It also normalized each activation by the activations of neighboring channels, a step that later networks dropped.

7.1.2.2 ReLU Activations

LeNet uses saturating activations: the sigmoid in our version, a scaled tanh in the original of 1998. Each sigmoid that the gradient passes on its way back to the first layer multiplies it by at most $1/4$, which is one reason why LeNet needed a learning rate as large as 2.0 (Section 6.6). AlexNet uses the ReLU, $\mathrm{ReLU}(x) = \max(x, 0)$ (Section 4.1.4), whose derivative is 1 for every positive input. A ReLU, unlike a sigmoid, therefore does not shrink the gradient on its active units. What remains is the scaling by the weights of each layer, which the initialization below is chosen to control. The ReLU is also cheaper to compute, since it needs no exponential. The difference is not subtle: on a small convolutional network for CIFAR-10, Krizhevsky et al. (2012) reached a training error of 25% six times faster with ReLUs than with tanh.

7.1.2.3 Dropout and Augmentation

The two fully connected layers of 4096 units hold most of the parameters and are where the model can memorize the training set. AlexNet applies dropout with probability 0.5 to their outputs (Section 4.6), whereas LeNet has no explicit regularizer at all. The crops and mirror images described above, together with random changes of color, add variation to the input.

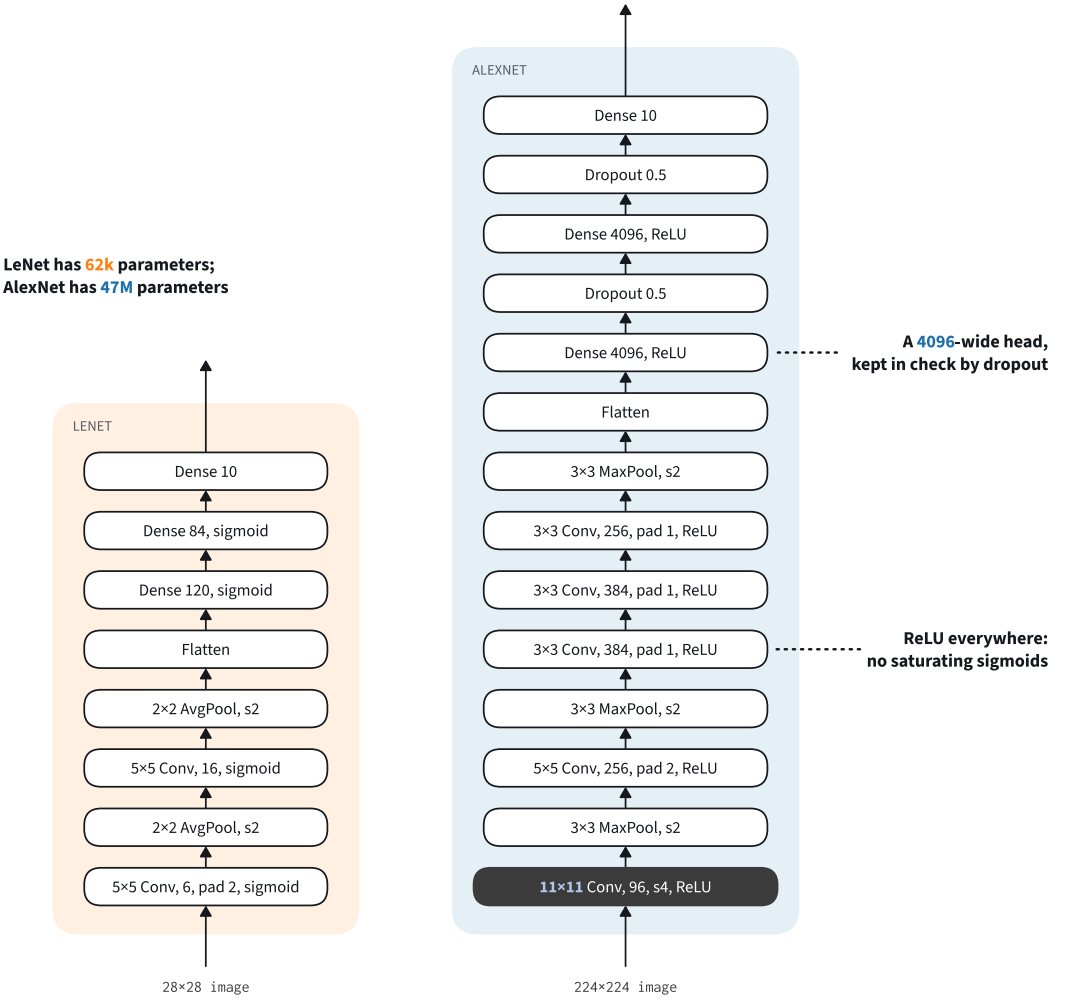


Figure 7.2: LeNet as built in Section 6.6 (left) and AlexNet as built in this section (right), both for single-channel Fashion-MNIST images and ten classes; the input is at the bottom. AlexNet is deeper and wider, uses ReLU activations and max-pooling, and regularizes its dense layers with dropout.

Augmentation has since grown into a large part of how convolutional networks are trained (Section 7.6).

7.1.2.4 Implementation

The code mirrors Figure 7.2 from bottom to top. We initialize the weights explicitly instead of relying on the library's defaults. LeNet's Xavier initialization keeps the variance of the activations constant through linear layers, but a ReLU sets about half of its inputs to zero and halves the second moment of what passes through it. *He initialization* (Section 4.4.2.2) compensates by doubling the variance. Every weight of a layer with n_{in} inputs per output is drawn from a normal distribution with mean zero and variance $2/n_{\text{in}}$, where $n_{\text{in}} = c_{\text{i}}k^2$ for a convolution. All biases start at zero, which is the default of every library here except PyTorch.

The lazy layers have no weights until the first minibatch fixes their shapes. As for LeNet, the `Trainer` then calls the model's `weight_init` on every layer.

```
class AlexNet(d2l.Classifier):
    def __init__(self, lr=0.1, num_steps=4690, num_classes=10):
        super().__init__()
        self.save_hyperparameters()
        self.net = nn.Sequential(
            nn.LazyConv2d(96, kernel_size=11, stride=4), nn.ReLU(),
            nn.MaxPool2d(kernel_size=3, stride=2),
            nn.LazyConv2d(256, kernel_size=5, padding=2), nn.ReLU(),
            nn.MaxPool2d(kernel_size=3, stride=2),
            nn.LazyConv2d(384, kernel_size=3, padding=1), nn.ReLU(),
            nn.LazyConv2d(384, kernel_size=3, padding=1), nn.ReLU(),
            nn.LazyConv2d(256, kernel_size=3, padding=1), nn.ReLU(),
            nn.MaxPool2d(kernel_size=3, stride=2), nn.Flatten(),
            nn.LazyLinear(4096), nn.ReLU(), nn.Dropout(p=0.5),
            nn.LazyLinear(4096), nn.ReLU(), nn.Dropout(p=0.5),
            nn.LazyLinear(num_classes))

    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')
            nn.init.zeros_(module.bias)
```

To check the code against Figure 7.2, we pass a single-channel 224×224 image through the network and print the output shape of each layer.

```
AlexNet().layer_summary((1, 1, 224, 224))
```

```
Conv2d                (1, 96, 54, 54)
ReLU                  (1, 96, 54, 54)
MaxPool2d             (1, 96, 26, 26)
Conv2d                (1, 256, 26, 26)
ReLU                  (1, 256, 26, 26)
MaxPool2d             (1, 256, 12, 12)
Conv2d                (1, 384, 12, 12)
```

```
ReLU                 (1, 384, 12, 12)
Conv2d               (1, 384, 12, 12)
ReLU                 (1, 384, 12, 12)
Conv2d               (1, 256, 12, 12)
ReLU                 (1, 256, 12, 12)
MaxPool2d            (1, 256, 5, 5)
Flatten              (1, 6400)
Linear               (1, 4096)
ReLU                 (1, 4096)
Dropout              (1, 4096)
Linear               (1, 4096)
ReLU                 (1, 4096)
Dropout              (1, 4096)
Linear               (1, 10)
```

The shapes follow the rules of Section 6.3. The first convolution maps 224 to $\lfloor (224 - 11)/4 \rfloor + 1 = 54$, and each 3×3 pooling with stride 2 maps n to $\lfloor (n - 3)/2 \rfloor + 1$: 54 to 26, 26 to 12, and 12 to 5. The padded convolutions in between keep the resolution, so the last convolutional stage hands $256 \times 5 \times 5 = 6400$ numbers to the first fully connected layer.

7.1.2.5 Parameters and Multiply-Adds

The shapes determine the size and the cost of every layer, counted as for LeNet (Section 6.6): a convolutional layer with c_i input channels, c_o output channels, and a $k \times k$ kernel applies each of its $c_\text{i} c_\text{o} k^2$ weights once at every output position, and a fully connected layer uses each weight once per example. Table 7.1 applies these counts to AlexNet.

Table 7.1: Parameters and multiply-adds per example of the layers of AlexNet, for one single-channel 224×224 image and ten classes. Pooling, activation, and dropout layers have no parameters.

layer	output	parameters	multiply-adds
11×11 convolution, 1 to 96, stride 4	96@54×54	11,712	33.9M
5×5 convolution, 96 to 256	256@26×26	614,656	415.3M
3×3 convolution, 256 to 384	384@12×12	885,120	127.4M
3×3 convolution, 384 to 384	384@12×12	1,327,488	191.1M
3×3 convolution, 384 to 256	256@12×12	884,992	127.4M
fully connected, 6400 to 4096	4096	26,218,496	26.2M
fully connected, 4096 to 4096	4096	16,781,312	16.8M
fully connected, 4096 to 10	10	40,970	0.04M
total		46,764,746	938.1M

The two kinds of layers divide the work as in LeNet, only more sharply. The convolutional layers hold 8% of the 47 million parameters and perform 95% of the multiply-adds. The fully connected layers hold the other 92% of the parameters, about 170 MB in 32-bit floating point, and perform 5% of the arithmetic. Against LeNet's 61,706 parameters and 416,520 multiply-adds per image, AlexNet has about 760 times the parameters and performs about 2,250 times the arithmetic. The network of the paper, with three input channels and 1000 classes, has about 60 million parameters, most of them in the same fully connected layers.

7.1.3 Training on Fashion-MNIST

Training AlexNet on ImageNet takes days even on a modern GPU, so we train it on Fashion-MNIST, as we did LeNet. Its 28×28 images are too small for an 11×11 kernel with stride 4, so we upsample them to 224×224 with the `resize` argument of `d2l.FashionMNIST`. Upsampling multiplies the number of pixels by 64 without adding information. We accept the waste in order to train the architecture as it is.

We train with minibatch SGD for 10 epochs, a third of LeNet's 30, with minibatches of 128 images. With ReLUs in place of sigmoids, a learning rate far below LeNet's 2.0 suffices.

7.1.3.1 Learning-Rate Schedule

A constant learning rate suits neither the first steps nor the last. At the start the weights are random, and a full-size step can throw the network into a region from which it recovers only slowly. At the end, a large rate keeps the weights jumping around a good solution instead of settling into it. AlexNet's authors took care of the end by hand: they divided the rate by 10 whenever the validation error stopped improving. We use a schedule that handles both ends without supervision. It is a later default and not part of AlexNet; the following sections use it too, so that their results can be compared. A linear *warmup* raises the rate from zero to its peak η_0 over the first t_w steps, and a half-cosine then lowers it to zero at the last step T:

$$\eta_t = \frac{\eta_0}{2}\left(1 + \cos\frac{\pi(t - t_\mathrm{w})}{T - t_\mathrm{w}}\right) \quad \text{for } t \geq t_\mathrm{w}. \tag{7.1}$$

At $t = t_\mathrm{w}$ the cosine is 1 and the rate is η_0; halfway through the decay it is $\eta_0/2$; at $t = T$ it is zero. Since the first steps are small, the peak can be higher than a rate that would be safe from the first step on. We train AlexNet at a peak of 0.03. How high the peak can go depends on the network, and Section 7.2 gives each of its networks the largest rate at which it trains reliably. The function `cosine_warmup` returns the factor by which the schedule multiplies the peak rate at step t; we save it in the `d2l` package for later sections. Section 9.6 compares learning-rate schedules in depth.

```
def cosine_warmup(t, num_steps, warmup_steps):
    """Learning-rate factor: linear warmup from 0, then a half-cosine to 0."""
    if t < warmup_steps:
        return t / warmup_steps
    progress = min((t - warmup_steps) / (num_steps - warmup_steps), 1)
    return 0.5 * (1 + math.cos(math.pi * progress))
```

`AlexNet` steps the schedule after every minibatch. Its argument `num_steps` is T, the number of minibatches in the whole run, and the warmup covers the first tenth of them: one epoch of the ten.

```
@d2l.add_to_class(AlexNet)
def configure_optimizers(self):
    optim = torch.optim.SGD(self.parameters(), lr=self.lr)
    schedule = torch.optim.lr_scheduler.LambdaLR(optim, lambda t: (
        cosine_warmup(t, self.num_steps, self.num_steps // 10)))
    return optim, schedule  # The Trainer steps it after every minibatch
```

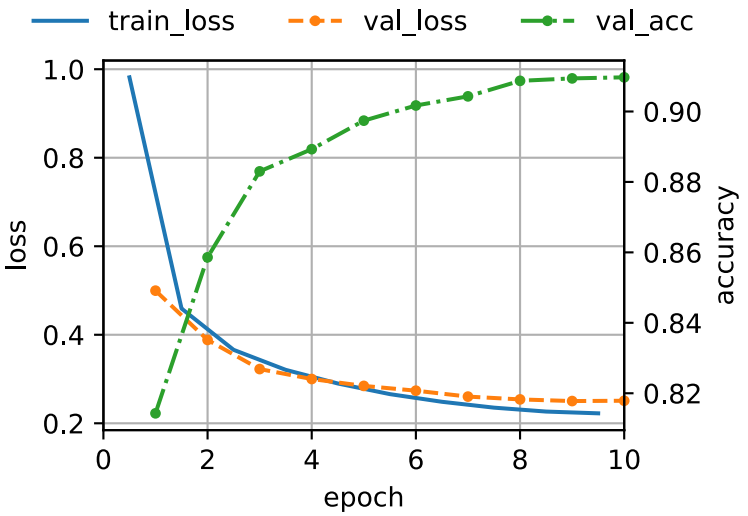
7.1.3.2 Comparison with LeNet

What should we expect? Fashion-MNIST has neither the number of images nor the detail that AlexNet's capacity is meant for. The run checks the implementation and measures the cost. It cannot show what made AlexNet win on ImageNet, and the gain over LeNet should be small. Before training we keep a copy of the 96 kernels of the first layer, so that we can see afterwards what training changed.

```
def first_kernels(model):
    """The 96 kernels of 11 x 11 weights of the first layer, as NumPy."""
    return d2l.numpy(model.net[0].weight)[:, 0]

data = d2l.FashionMNIST(batch_size=128, resize=(224, 224))
model = AlexNet(lr=0.03, num_steps=10 * len(data.train_dataloader()))
model.apply_init([next(iter(data.train_dataloader()))[0][:1]],
                 model.weight_init)  # Create the weights now
K0 = first_kernels(model)
```

```
trainer = d2l.Trainer(max_epochs=10, num_gpus=1)
trainer.fit(model, data)
```



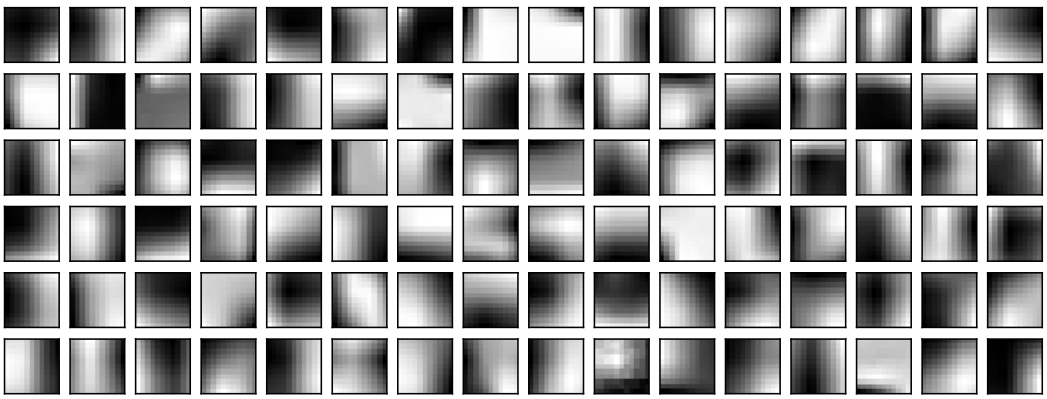
AlexNet reaches a validation accuracy of about 91% after 10 epochs, with a training loss of about 0.22 and a validation loss of about 0.25. Over the last epochs, as the learning rate approaches zero, the accuracy changes by less than a point from one epoch to the next, so the last epoch is a fair summary of the run. LeNet reaches about 88–90% after 30 epochs. The larger network thus passes LeNet's accuracy in a third of the epochs, but only by a point or two, and it performs 2,250 times the arithmetic per image to do so. Nor can we credit its size alone with even this gain, since the two runs differ in activation, initialization, and schedule as well.

7.1.3.3 The Learned First Layer

LeNet's six kernels of 5×5 weights were too small to tell the eye much (Section 6.6). AlexNet's 96 kernels of 11×11 weights are large enough to read as pictures, and Figure 7.1 showed what they become on ImageNet. What did ours learn from Fashion-MNIST? We subtract the copy taken before training from the kernels after it and draw the difference, each kernel on its own gray scale.

```
change = first_kernels(model) - K0
d2l.show_images(change, 6, 16, scale=0.6, cmap='gray')
print(f'mean size of the change: {np.abs(change).mean() / np.abs(K0).mean():.3f}'
      ' of the mean size of the initial weights')
```

```
mean size of the change: 0.042 of the mean size of the initial weights
```



Ten epochs moved the kernels by only about 4% of the size of their random initial values, so a trained kernel still looks like the noise it started from. The change itself is not noise. It is smooth: brightness ramps and broad bars at various orientations, the low-frequency relatives of the edge detectors of Figure 7.1. The sharp edges and the color blobs of that figure are missing, and the input explains both. Our images were upsampled eightfold, so they contain no detail finer than about eight pixels, which an 11×11 kernel can only see as a gradual change. And they have a single channel. The first layer learned what the data contain.

Most of AlexNet's capacity, as Table 7.1 showed, sits in a fully connected head that does little of the arithmetic and needs dropout to keep it from memorizing the training set. The next section asks whether a network needs that head at all.

7.1.4 Summary

Learned features overtook hand-designed ones when three things were in place at once: a dataset large enough to constrain tens of millions of parameters, GPUs that could compute their gradients in days, and ReLU activations, dropout, and data augmentation to keep such a network trainable. With them AlexNet, LeNet's template made deeper and wider, cut the ImageNet top-5 error from 26% to 15%. Counting its layers shows that the convolutions perform 95% of the arithmetic while the fully connected head holds 92% of the parameters. On Fashion-MNIST, which cannot use that capacity, AlexNet passes LeNet's accuracy by a point or two in a third of the epochs, at 2,250 times the arithmetic per image.

7.1.5 Exercises

1. **Memory and compute ledger.** Extend Table 7.1 to the memory that `AlexNet` needs in 32-bit floating point.

    1. Compute the number of activations that the forward pass of one 224×224 image stores, layer by layer, counting the output of every convolutional, pooling, and fully connected layer. Where is the memory for activations concentrated, and where is the memory for parameters concentrated?
    2. During training, the parameters, their gradients, and the stored activations of a whole minibatch must fit in memory. At which batch size does the memory for activations exceed the memory for parameters and gradients?
    3. Repeat the parameter count of Table 7.1 for the ImageNet version, with three input channels, 1000 classes, and 227×227 inputs, which make the last convolutional stage 6×6 instead of 5×5. Compare your total with the 60 million parameters reported by Krizhevsky et al. (2012) .

    Adapted from Simon Prince, Understanding Deep Learning, Problem 10.16.

2. **Receptive fields of the early layers.** A stack of layers with kernel sizes k_i and strides s_i gives its top unit a receptive field of side $r = 1 + \sum_i (k_i - 1) \prod_{j<i} s_j$ (Section 6.3). Apply it to `AlexNet`, counting each max-pooling layer as a layer with its own kernel size and stride, to compute the receptive field of one unit after each of the five convolutional layers. What fraction of the 224×224 input side does each receptive field span, and where does a unit first see the whole image?

    Adapted from Simon Prince, Understanding Deep Learning, Problem 10.17.

3. code **Change per layer.** Training moved the first-layer kernels by about 4% of the size of their initial values.

    1. Keep a copy of every weight array before training and compute the same ratio for each of the eight layers with weights. Which layers moved most, and which least? Relate the order to the number of weights in each layer.
    2. The text attributes the smoothness of the first layer's change to the eightfold upsampling. Test this: train on images resized to 112×112, which are upsampled fourfold (the last convolutional stage is then 2×2, so the first fully connected layer has 1024 inputs), and draw the change of the first layer again. At which scale does it vary now?

4. code **Dropout.** The fully connected layers of `AlexNet` hold 92% of its parameters, and dropout is its only regularizer.

1. Train `AlexNet` as in this section with both dropout layers removed, and compare the training loss, the validation loss, and the validation accuracy after 10 epochs with those of the original.
2. Train both networks on a random subset of 6,000 training images for the same number of gradient steps. How do the differences change, and why?

5. **Arithmetic intensity.** ▯ The *arithmetic intensity* of a layer is the number of multiply-adds it performs per byte of parameters it reads from memory. Use Table 7.1.
   1. Compute the arithmetic intensity of every layer of `AlexNet`, once for a batch of 1 image and once for a batch of 128 images, assuming that each parameter is read once per batch.
   2. Consider a chip that performs 10 multiply-adds in the time it reads one byte from memory. Assume that each layer takes the longer of its arithmetic time and its parameter-reading time. A designer can spend extra chip area on doubling either the arithmetic throughput or the memory bandwidth. Which choice speeds up single-image inference more, and which speeds up training at batch size 128 more?

6. code **Batch size, throughput, and memory.** ▯ Train `AlexNet` as in this section while varying only the batch size.
   1. Using your answer to problem 1, predict the largest training batch size that fits in the memory of your GPU.
   2. Starting from a batch size of 32 and doubling it, find the largest batch size that trains without running out of memory. Compare it with your prediction and name the main source of any discrepancy.
   3. Measure the throughput in images per second over a fixed number of training steps for at least four batch sizes up to that limit. Beyond which batch size does throughput stop growing, and why?

7.2 Blocks, Bottlenecks, and Branches: VGG, NiN, GoogLeNet

AlexNet (Section 7.1) specifies each of its eight layers by hand: an 11×11 kernel here, 384 channels there, a pooling layer after the first, second, and fifth convolutions. That works for eight layers, but a designer who wants twenty needs a rule for choosing them. AlexNet also puts 92% of its 47 million parameters into its fully connected head, where they do 5% of the arithmetic and have to be held back from memorizing the training set by dropout. Two networks of 2013 and 2014 each solved one of these problems, and a third combined their answers and added a question of its own. Nobody trains these networks any more, but their ideas are in every convolutional network since.

VGG (Simonyan and Zisserman 2015) builds the network from one repeated *block*, a few 3×3 convolutions followed by pooling, so that a whole network is a short list of block sizes. Network in network (NiN) (Lin et al. 2013) removes the fully connected head: 1×1 convolutions mix the channels at every position, and an average over all positions turns the last feature map into class scores. GoogLeNet (Szegedy et al. 2015) asks which kernel size a layer should have and declines to choose: it runs convolutions of several sizes in parallel inside one block and makes the expensive ones affordable by first reducing the number of channels with 1×1 convolutions. We build each network, count its parameters and multiply-

adds as we did for AlexNet, and train it on Fashion-MNIST under the same schedule, so that the four networks can be compared in one table.

7.2.1 VGG: Blocks as the Unit of Design

VGG replaces individually chosen layers with one block, repeated at growing channel counts. The block answers two questions at once: how to make a network deeper without losing its resolution too soon, and how large its kernels should be.

7.2.1.1 Stacks of 3×3 Convolutions

If every convolution were followed by pooling with stride 2, an input of width d would permit at most $\lfloor \log_2 d \rfloor$ convolutions before its spatial extent collapses: for a 224-pixel image, 224, 112, 56, 28, 14, 7, 3, 1, and the network is over after seven halvings. To go deeper, a network must place several convolutions at each resolution, with stride 1 and padding that keeps the size.

Which kernels should it place there? VGG's answer is the smallest one that still sees a neighbor. Two stacked 3×3 convolutions see a 5×5 window of their input, the same as one 5×5 convolution, but with c input and c output channels they have $2 \cdot 9c^2 = 18c^2$ weights instead of $25c^2$. Three of them see 7×7 with $27c^2$ weights instead of $49c^2$. The stack is not only cheaper. It applies three nonlinearities where the single large kernel applies one, so it can compute functions of its 7×7 window that one linear filter followed by a ReLU cannot. Stacks of 3×3 convolutions remained the default for most of the following decade.

A *VGG block* is therefore a sequence of `num_convs` 3×3 convolutions with padding 1, each followed by a ReLU, and a 2×2 max-pooling with stride 2 at the end. The convolutions keep the height and the width; the pooling halves them.

```
def vgg_block(num_convs, out_channels):
    layers = []
    for _ in range(num_convs):
        layers.append(nn.LazyConv2d(out_channels, kernel_size=3, padding=1))
        layers.append(nn.ReLU())
    layers.append(nn.MaxPool2d(kernel_size=2, stride=2))
    return nn.Sequential(*layers)
```

7.2.1.2 The VGG Network

A VGG network is a sequence of such blocks followed by the fully connected head of AlexNet: two hidden layers of 4096 units with dropout, and an output layer. Figure 7.3 places VGG-11 beside AlexNet. Every block halves the resolution, so five blocks take a 224×224 input to 7×7, and the number of channels doubles from block to block up to 512.

Since a block is fixed by two numbers, the number of convolutions and their output channels, the convolutional part of the network is fixed by a list of such pairs. The class `VGG` takes this list as its argument `arch` and builds one block per entry. VGG is therefore a *family* of networks: VGG-11, VGG-16, and VGG-19 differ only in `arch`. The network has become a piece of data, and this, more than any layer, is what VGG left behind. We initialize all networks of this section as AlexNet in Section 7.1: He-normal weights and zero biases.

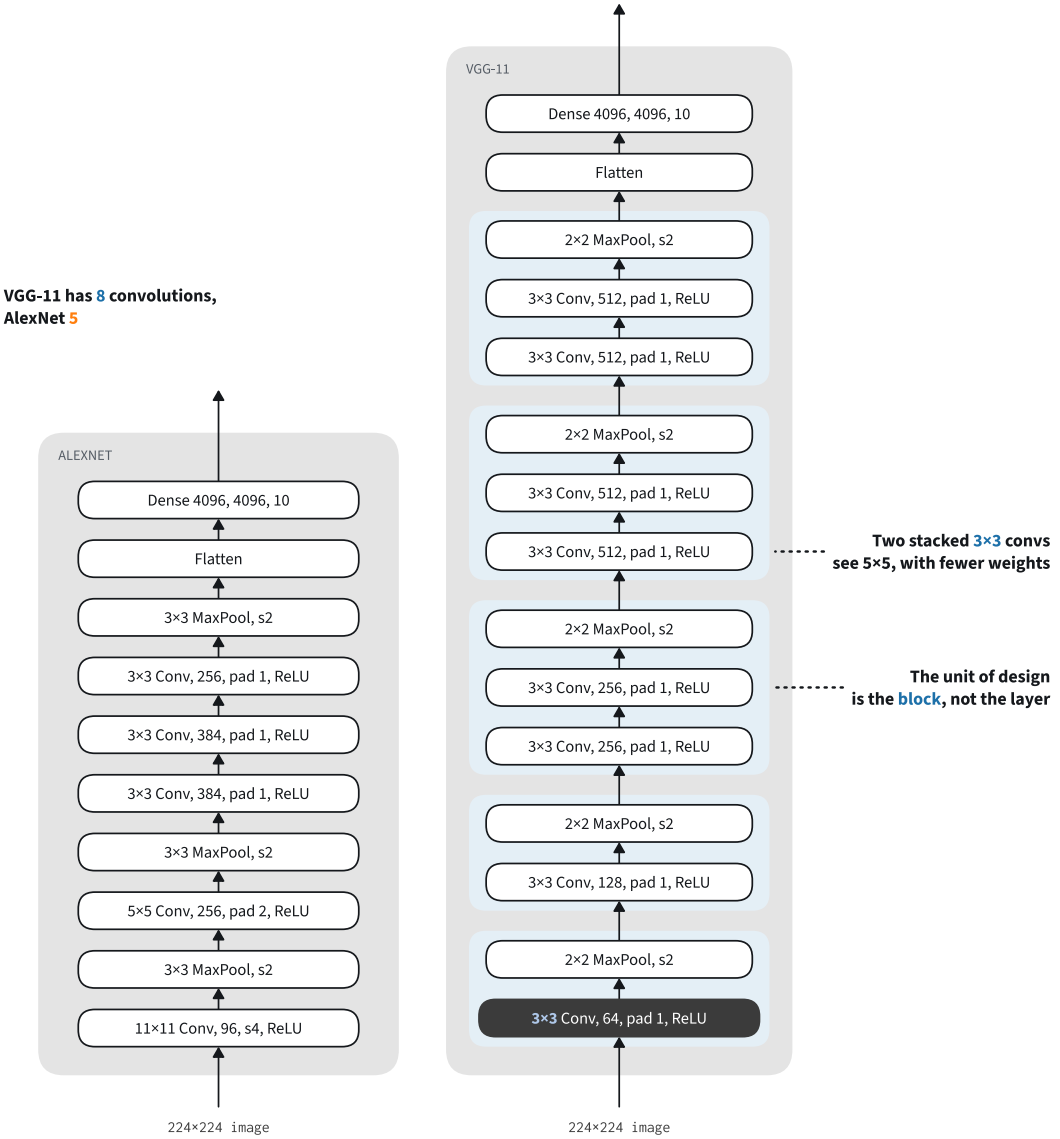


Figure 7.3: AlexNet (left) and VGG-11 (right) for single-channel 224×224 images and ten classes; the input is at the bottom. AlexNet's layers are chosen one by one; VGG repeats one block, several 3×3 convolutions followed by max-pooling, at growing channel counts.

As for AlexNet, the `Trainer` calls `weight_init` on every layer once the first minibatch has fixed the shapes of the lazy layers.

```
class VGG(d2l.Classifier):
    def __init__(self, arch, lr=0.1, num_steps=4690, num_classes=10):
        super().__init__()
        self.save_hyperparameters()
        conv_blks = [vgg_block(num_convs, out_channels)
                     for (num_convs, out_channels) in arch]
        self.net = nn.Sequential(*conv_blks, *self.head())

    def head(self):  # AlexNet's fully connected head
        return [nn.Flatten(),
                nn.LazyLinear(4096), nn.ReLU(), nn.Dropout(0.5),
                nn.LazyLinear(4096), nn.ReLU(), nn.Dropout(0.5),
                nn.LazyLinear(self.num_classes)]

    def weight_init(self, module):  # He-normal weights, zero biases
        if type(module) in (nn.Linear, nn.Conv2d):
            nn.init.kaiming_normal_(module.weight, nonlinearity='relu')
            nn.init.zeros_(module.bias)
```

VGG-11, the smallest network of the paper, has five blocks: one convolution each in the first two and two in the other three, with 64, 128, 256, 512, and 512 output channels. Its eight convolutional and three fully connected layers give it its name. We check the shapes on one 224×224 image.

```
VGG(arch=((1, 64), (1, 128), (2, 256), (2, 512), (2, 512))).layer_summary(
    (1, 1, 224, 224))
```

```
Sequential            (1, 64, 112, 112)
Sequential            (1, 128, 56, 56)
Sequential            (1, 256, 28, 28)
Sequential            (1, 512, 14, 14)
Sequential            (1, 512, 7, 7)
Flatten               (1, 25088)
Linear                (1, 4096)
ReLU                  (1, 4096)
Dropout               (1, 4096)
Linear                (1, 4096)
ReLU                  (1, 4096)
Dropout               (1, 4096)
Linear                (1, 10)
```

The last block hands $512 \times 7 \times 7 = 25{,}088$ numbers to the fully connected head. Counted as for AlexNet, VGG-11 has 128.8 million parameters and performs 7.5 billion multiply-adds per image, eight times as many as AlexNet. The parameters, once again, are not where the arithmetic is: the head holds 93% of them, and its first fully connected layer alone, $25{,}088 \times 4096$, holds 103 million.

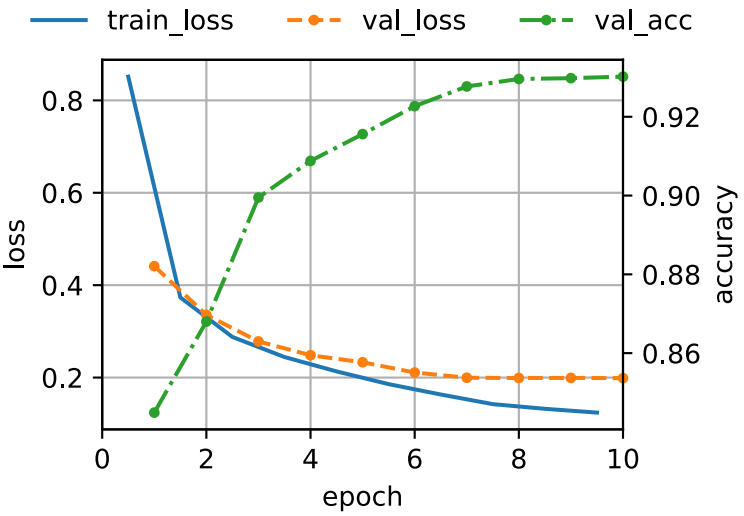
7.2.1.3 Training

Eight times AlexNet's arithmetic is more than a demonstration should cost, so we train a narrower member of the family on Fashion-MNIST, with a quarter of the channels in every block: 16, 32, 64, 128, and 128. It performs 512 million multiply-adds per image, about half of AlexNet's 938 million. Everything else follows Section 7.1: images upsampled to 224×224, minibatches of 128, and 10 epochs of SGD under the warmup and cosine schedule Equation 7.1. VGG gets the same `configure_optimizers` as AlexNet.

Only the peak learning rate differs. At a peak of 0.1 some networks of this section are at the edge of stability: in one run or another the loss diverges, or the network never leaves chance. Such a network trains at 0.03. VGG is not among them. It trains reliably at 0.1 and ends about a point and a half higher there than at 0.03, so it trains at 0.1.

```
@d2l.add_to_class(VGG)
def configure_optimizers(self):
    optim = torch.optim.SGD(self.parameters(), lr=self.lr)
    schedule = torch.optim.lr_scheduler.LambdaLR(optim, lambda t: (
        d2l.cosine_warmup(t, self.num_steps, self.num_steps // 10)))
    return optim, schedule  # The Trainer steps it after every minibatch
```

```
data = d2l.FashionMNIST(batch_size=128, resize=(224, 224))
model = VGG(arch=((1, 16), (1, 32), (2, 64), (2, 128), (2, 128)), lr=0.1,
            num_steps=10 * len(data.train_dataloader()))
trainer = d2l.Trainer(max_epochs=10, num_gpus=1)
trainer.fit(model, data)
val_acc = {'VGG': trainer.history['val_acc']}  # for the comparison below
```



The narrow VGG reaches a validation accuracy of about 93%, two points above AlexNet with half of its arithmetic. Most of that margin, to be fair, is the higher peak: at AlexNet's 0.03 the same network ends just below 92%, as we will see below. Note, however, what the narrowing did not change. The convolutions shrank to 0.6 million parameters, but the

head still maps $128 \times 7 \times 7 = 6272$ numbers to 4096 units and holds 42.5 million of the network's 43.1 million parameters: 99%. Blocks organized the body of the network; the head is untouched.

7.2.2 NiN: 1×1 Convolutions and Global Average Pooling

The fully connected head is expensive because it connects every position of the last feature map to every hidden unit. All those connections serve one step, from a feature map to one score per class, and it is fair to ask whether that step needs 40 million parameters. NiN removes the head altogether, and it changes the body as well. A convolution followed by a ReLU computes, at each position, a thresholded linear function of its window. To compute something richer from the same window without widening it, we can add layers that look at one position only. Luckily, we have already met such a layer (Section 6.4.3).

7.2.2.1 NiN Blocks

A 1×1 convolution is a fully connected layer applied at every pixel, with the same weights at every position. A *NiN block* follows an ordinary convolution with two 1×1 convolutions, each with a ReLU. The two together form a small MLP with one hidden layer that runs at every pixel, hence the name *network in network*. Figure 7.4 compares the head of VGG with the end of NiN.

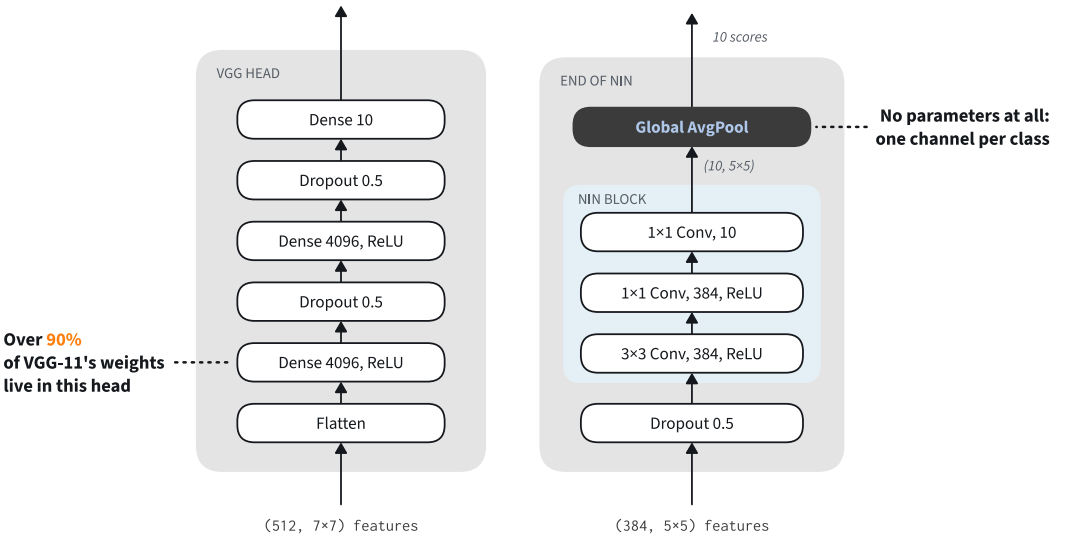


Figure 7.4: The head of VGG (left) and the end of NiN (right), both for ten classes; the input is at the bottom. VGG flattens its last feature map into fully connected layers. After dropout, NiN's last block, a 3×3 convolution and two 1×1 convolutions with no ReLU after the last one, emits one channel per class, and global average pooling turns each channel into one score.

```
def nin_block(out_channels, kernel_size, strides, padding, num_classes=None):
    layers = [
        nn.LazyConv2d(out_channels, kernel_size, strides, padding), nn.ReLU(),
```

```
        nn.LazyConv2d(out_channels, kernel_size=1), nn.ReLU()]
    if num_classes is None:
        layers += [nn.LazyConv2d(out_channels, kernel_size=1), nn.ReLU()]
    else:  # Last block: one map of scores per class, no ReLU
        layers.append(nn.LazyConv2d(num_classes, kernel_size=1))
    return nn.Sequential(*layers)
```

7.2.2.2 Global Average Pooling

NiN uses the kernel sizes and channel counts of AlexNet's first layers: blocks with 11×11, 5×5, and 3×3 convolutions and 96, 256, and 384 channels, each followed by 3×3 max-pooling with stride 2. Then it differs. Its last block, again with 384 channels, ends in a 1×1 convolution with as many output channels as there are classes, ten for Fashion-MNIST, and *global average pooling* averages each of these channels over all positions. The ten averages are the logits. The head therefore has no parameters at all: the last block must learn to make the channel of the correct class large on average over the image.

This last 1×1 convolution outputs the class scores directly, without a ReLU; `nin_block` builds it when it is given `num_classes`. A ReLU there would switch a class off for good: once a class's pre-activations are negative at every position of every image, its logit is 0, its gradient is zero, and training can never bring it back.

Global average pooling has a second property. A convolution commutes with translations, so shifting the input shifts each feature map, and an average over all positions does not change when its entries are permuted. Up to the borders of the image and the strides of the network, the scores therefore do not depend on where in the image an object appears. A fully connected head, by contrast, has a separate weight for every position.

```
class NiN(d2l.Classifier):
    def __init__(self, lr=0.1, num_steps=4690, num_classes=10):
        super().__init__()
        self.save_hyperparameters()
        self.net = nn.Sequential(
            nin_block(96, kernel_size=11, strides=4, padding=0),
            nn.MaxPool2d(3, stride=2),
            nin_block(256, kernel_size=5, strides=1, padding=2),
            nn.MaxPool2d(3, stride=2),
            nin_block(384, kernel_size=3, strides=1, padding=1),
            nn.MaxPool2d(3, stride=2),
            nn.Dropout(0.5),
            nin_block(384, kernel_size=3, strides=1, padding=1,
                      num_classes=num_classes),
            nn.AdaptiveAvgPool2d((1, 1)),
            nn.Flatten())

    weight_init = VGG.weight_init  # He-normal weights, zero biases
```

We check the shapes on one 224×224 image.

```
NiN().layer_summary((1, 1, 224, 224))
```

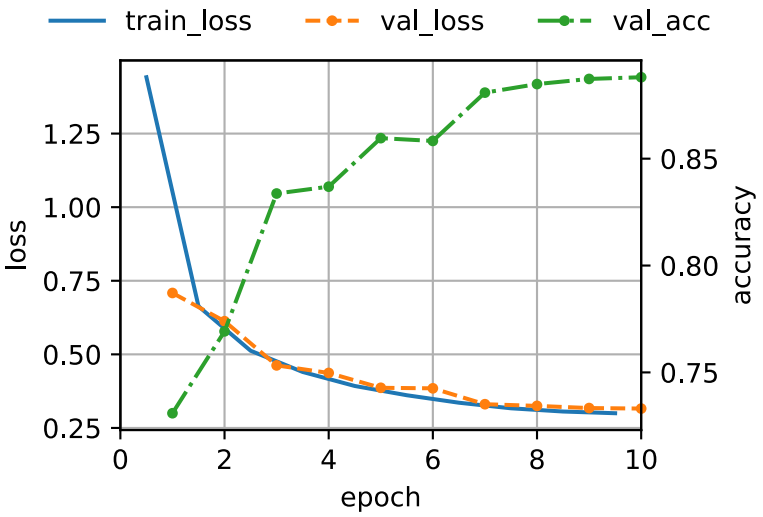
```
Sequential              (1, 96, 54, 54)
MaxPool2d               (1, 96, 26, 26)
Sequential              (1, 256, 26, 26)
MaxPool2d               (1, 256, 12, 12)
Sequential              (1, 384, 12, 12)
MaxPool2d               (1, 384, 5, 5)
Dropout                 (1, 384, 5, 5)
Sequential              (1, 10, 5, 5)
AdaptiveAvgPool2d       (1, 10, 1, 1)
Flatten                 (1, 10)
```

The resolution falls from 224 to 54, 26, 12, and 5, as in AlexNet, and the last block emits ten channels of 5×5, which the pooling reduces to ten numbers. NiN has 3.4 million parameters, 7% of AlexNet's, and performs 798 million multiply-adds per image, somewhat fewer than AlexNet's 938 million: the 1×1 convolutions add arithmetic to the body while the head disappears.

7.2.2.3 Training

We train NiN as we trained VGG, with VGG's `configure_optimizers`, but at a peak of 0.03. NiN is one of the networks for which 0.1 is too close to the edge: some runs diverge there.

```
NiN.configure_optimizers = VGG.configure_optimizers  # AlexNet's schedule
data = d2l.FashionMNIST(batch_size=128, resize=(224, 224))
model = NiN(lr=0.03, num_steps=10 * len(data.train_dataloader()))
trainer = d2l.Trainer(max_epochs=10, num_gpus=1)
trainer.fit(model, data)
val_acc['NiN'] = trainer.history['val_acc']
```



NiN ends at a validation accuracy of about 89% after the same 10 epochs, two points below AlexNet and four below VGG. Its training loss ends near 0.3, against about 0.22 for AlexNet, and its validation loss stays close to it: in 10 epochs NiN fits the training data less well, and it does not overfit. This is a rather modest result for a network that performs nearly as much

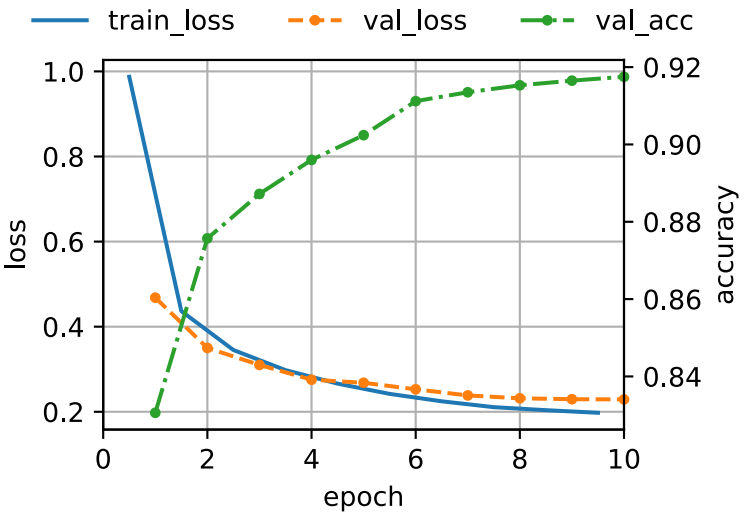
arithmetic as AlexNet. NiN is remembered for its two ideas, not for its accuracy: 1×1 convolutions for mixing channels, and global average pooling as the head of a convolutional network.

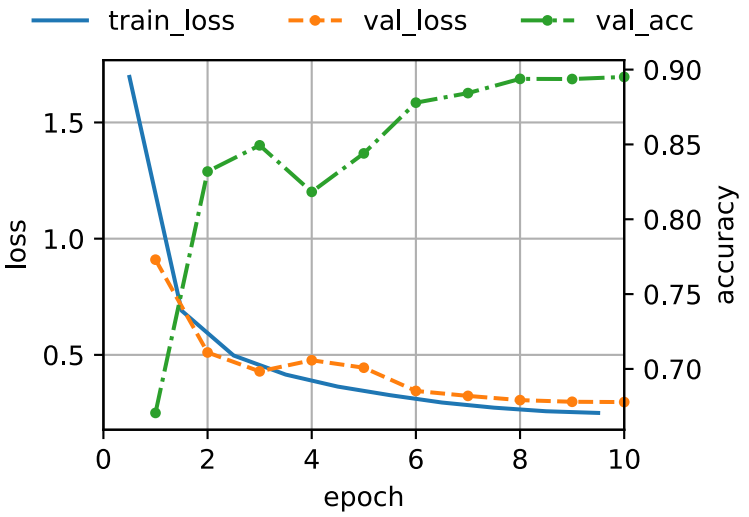
7.2.2.4 A Pooled Head for VGG

NiN changes the body and the head at once, so its result cannot tell us what a dense head contributes. A clean test changes the head and nothing else. `VGG` builds its head in a method of its own, `head`, so a subclass can replace it. `PooledVGG` keeps the quarter-width body of VGG and ends it with global average pooling over the 128 channels of the last block and one fully connected layer that maps them to the ten classes. This removes 42.5 million of the network's 43.1 million parameters, and as many multiply-adds. The pooled head, like NiN, diverges or collapses in some runs at 0.1. Since the comparison should measure the head and not the learning rate, we train both networks at the same peak, 0.03.

```
class PooledVGG(VGG):
    def head(self):  # Global average pooling and one fully connected layer
        return [nn.AdaptiveAvgPool2d((1, 1)), nn.Flatten(),
                nn.LazyLinear(self.num_classes)]
```

```
heads = {}  # Both heads at peak 0.03, the rate the pooled head tolerates
for net in (VGG, PooledVGG):
    model = net(arch=((1, 16), (1, 32), (2, 64), (2, 128), (2, 128)),
                lr=0.03, num_steps=10 * len(data.train_dataloader()))
    trainer = d2l.Trainer(max_epochs=10, num_gpus=1)
    trainer.fit(model, data)
    heads[net.__name__] = round(trainer.history['val_acc'][-1], 3)
val_acc['PooledVGG'] = trainer.history['val_acc']
heads
```






```
{'VGG': 0.917, 'PooledVGG': 0.895}
```

The pooled VGG ends at 89–90%, about two points below the same body with its dense head, which reaches just under 92% at this peak. Its training loss, about 0.25, stays above the dense network's 0.2. So the head does matter. On Fashion-MNIST, for this narrow body and 10 epochs, the 42 million parameters of the dense head are worth about two points.

7.2.3 GoogLeNet: Multi-Branch Blocks

Which kernel size should a layer use? A 1×1 convolution sees one position, a 5×5 convolution sees a neighborhood, and objects in photographs come at every size. GoogLeNet does not choose: its *Inception block* applies several kernel sizes in parallel to the same input and concatenates their outputs. Of course, three kernel sizes cost more than one. To keep the large kernels affordable, GoogLeNet borrows NiN's 1×1 convolution and uses it to reduce the number of channels before each of them.

7.2.3.1 The Inception Block

The Inception block of Figure 7.5 has four parallel branches. The first applies a 1×1 convolution. The second and third apply a 1×1 convolution that reduces the number of channels, followed by a 3×3 or a 5×5 convolution. The fourth applies 3×3 max-pooling with stride 1, followed by a 1×1 convolution. Every branch pads its input so that its output keeps the height and the width, and the block concatenates the four outputs along the channel dimension. The numbers of output channels of the branches are the block's hyperparameters: they decide how much of the block's capacity goes to each kernel size. The name alludes to the film *Inception* and the catchphrase it inspired, "we need to go deeper".

```
class Inception(nn.Module):
    # c1--c4 are the numbers of output channels of the four branches
    def __init__(self, c1, c2, c3, c4):
        super().__init__()
        self.b1_1 = nn.LazyConv2d(c1, kernel_size=1)
```

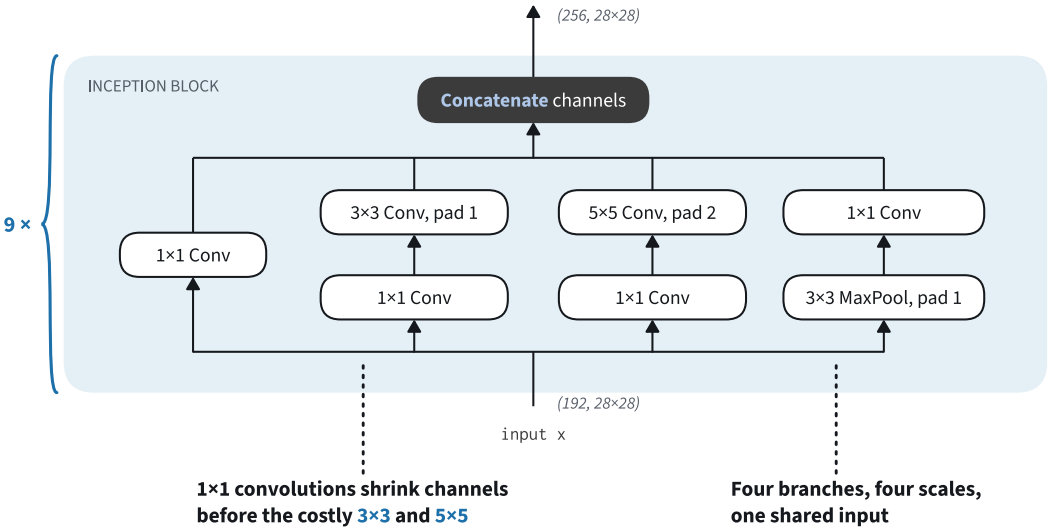


Figure 7.5: The Inception block; the input is at the bottom. Four parallel branches read the same input with different kernel sizes; 1×1 convolutions reduce the number of channels before the 3×3 and 5×5 convolutions; the branch outputs are concatenated along the channel dimension. The shapes are those of the first Inception block of GoogLeNet on a 224×224 input.

```
        self.b2_1 = nn.LazyConv2d(c2[0], kernel_size=1)
        self.b2_2 = nn.LazyConv2d(c2[1], kernel_size=3, padding=1)
        self.b3_1 = nn.LazyConv2d(c3[0], kernel_size=1)
        self.b3_2 = nn.LazyConv2d(c3[1], kernel_size=5, padding=2)
        self.b4_1 = nn.MaxPool2d(kernel_size=3, stride=1, padding=1)
        self.b4_2 = nn.LazyConv2d(c4, kernel_size=1)

    def forward(self, x):
        b1 = F.relu(self.b1_1(x))
        b2 = F.relu(self.b2_2(F.relu(self.b2_1(x))))
        b3 = F.relu(self.b3_2(F.relu(self.b3_1(x))))
        b4 = F.relu(self.b4_2(self.b4_1(x)))
        return torch.cat((b1, b2, b3, b4), dim=1)
```

The first Inception block of GoogLeNet shows the role of the 1×1 reductions. On a 224×224 input it receives 192 channels at 28×28 resolution, the shapes annotated in Figure 7.5. With branch outputs $c_1 = 64$, $c_2 = (96, 128)$, $c_3 = (16, 32)$, and $c_4 = 32$, it emits $64 + 128 + 32 + 32 = 256$ channels at the same 28×28 resolution. A 5×5 convolution straight from 192 to 32 channels would need $25 \cdot 192 \cdot 32 = 153{,}600$ weights. Reducing to 16 channels first costs $192 \cdot 16 = 3{,}072$ weights for the reduction and $25 \cdot 16 \cdot 32 = 12{,}800$ for the convolution, 15,872 in total: a tenth. The number of multiply-adds falls by the same factor, since every weight is used at all 28×28 positions. A narrow layer placed before an expensive one in this way is called a *bottleneck*. It is the most widely reused idea of this section, and we will meet it again in Section 7.4.5.1 and Section 7.5.

7.2.3.2 Stem, Body, and Head

GoogLeNet has three parts, an organization that most later vision networks share. The *stem* processes the raw image with a few ordinary convolutions and reduces its resolution; here it has five layers, three convolutions and two max-poolings, which shrink the image eight-fold and emit 192 channels. The *body* consists of nine Inception blocks in three groups of two, five, and two, with max-pooling between the groups. The *head* is NiN's: global average pooling, followed by a single fully connected layer that maps the channels to the classes.

The channel allocations of the nine blocks come from the paper, which chose them by hand, and there is no rule to be learned from them. As for VGG, we store them as data, one tuple per block and one tuple of blocks per group:

```
googlenet_arch = (
    ((64, (96, 128), (16, 32), 32), (128, (128, 192), (32, 96), 64)),
    ((192, (96, 208), (16, 48), 64), (160, (112, 224), (24, 64), 64),
     (128, (128, 256), (24, 64), 64), (112, (144, 288), (32, 64), 64),
     (256, (160, 320), (32, 128), 128)),
    ((256, (160, 320), (32, 128), 128), (384, (192, 384), (48, 128), 128)))
```

The class `GoogLeNet` takes this tuple as its argument `arch`, as `VGG` does, and composes the stem, the three groups with pooling in between, and the head.

```
class GoogLeNet(d2l.Classifier):
    def __init__(self, arch=googlenet_arch, lr=0.1, num_steps=4690,
                 num_classes=10):
        super().__init__()
        self.save_hyperparameters()
        pool = lambda: nn.MaxPool2d(kernel_size=3, stride=2, padding=1)
        stem = nn.Sequential(
            nn.LazyConv2d(64, kernel_size=7, stride=2, padding=3), nn.ReLU(),
            pool(),
            nn.LazyConv2d(64, kernel_size=1), nn.ReLU(),
            nn.LazyConv2d(192, kernel_size=3, padding=1), nn.ReLU(), pool())
        body = [nn.Sequential(*[Inception(*c) for c in group])
                for group in arch]
        head = nn.Sequential(nn.AdaptiveAvgPool2d((1, 1)), nn.Flatten(),
                             nn.LazyLinear(num_classes))
        self.net = nn.Sequential(stem, body[0], pool(), body[1], pool(),
                                 body[2], head)

    weight_init = VGG.weight_init  # He-normal weights, zero biases
```

GoogLeNet has five stride-2 steps, three in the stem (the first convolution and two poolings) and two between the groups, so it reduces the resolution by a factor of 32. On a 96×96 input, a resolution at which Fashion-MNIST training stays cheap, the groups operate at 12×12, 6×6, and 3×3; the number of channels grows from 192 after the stem to 480, 832, and 1024 after the three groups.

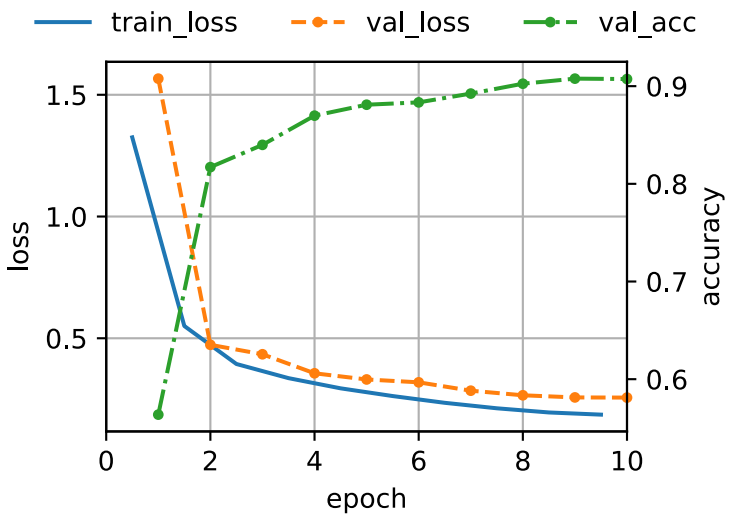
```
GoogLeNet().layer_summary((1, 1, 96, 96))
```

```
Sequential            (1, 192, 12, 12)
```

```
Sequential              (1, 480, 12, 12)
MaxPool2d               (1, 480, 6, 6)
Sequential              (1, 832, 6, 6)
MaxPool2d               (1, 832, 3, 3)
Sequential              (1, 1024, 3, 3)
Sequential              (1, 10)
```

GoogLeNet is the deepest network of this section: 22 layers with weights lie on its longest path, three in the stem, two in each of the nine Inception blocks, and the output layer. We train it as NiN, at a peak of 0.03, because at 0.1 an occasional run never leaves chance. One difference will matter when we compare the networks: the images are 96×96, not 224×224. At this size GoogLeNet's 6.0 million parameters perform 276 million multiply-adds per image.

```
GoogLeNet.configure_optimizers = VGG.configure_optimizers
data = d2l.FashionMNIST(batch_size=128, resize=(96, 96))
model = GoogLeNet(lr=0.03, num_steps=10 * len(data.train_dataloader()))
trainer = d2l.Trainer(max_epochs=10, num_gpus=1)
trainer.fit(model, data)
val_acc['GoogLeNet'] = trainer.history['val_acc']
```



GoogLeNet ends at a validation accuracy of about 91% after 10 epochs, the same as AlexNet at the same peak. Its training loss, just below 0.2, is a little lower than AlexNet's, and its validation loss, about 0.25, is the same.

7.2.4 Comparing the Networks

Since the learning rate decays to zero, the validation accuracy of every network changes little over the last epochs, and we compare the networks by their accuracy after the last epoch:

```
{name: round(acc[-1], 3) for name, acc in val_acc.items()}  # After 10 epochs
```

```
{'VGG': 0.93, 'NiN': 0.888, 'PooledVGG': 0.895, 'GoogLeNet': 0.907}
```

Table 7.2 collects these accuracies for the networks of this section and AlexNet, all trained on Fashion-MNIST for 10 epochs with the same schedule, each at its own peak rate. It adds LeNet and the counts of parameters and multiply-adds at the input size used for training.

Table 7.2: Parameters, multiply-adds, and Fashion-MNIST validation accuracy of the networks of Section 6.6, Section 7.1, and this section. Multiply-adds are per image at the training input size. Accuracies are after 10 epochs at the peak learning rate given, as ranges over repeated runs; a single value means that the runs agree to within half a point. LeNet's are after 30 epochs at a constant learning rate.

network	input	parameters	multiply-adds	peak rate	validation accuracy
LeNet	28×28	0.06M	0.4M	2.0	88–90% (30 epochs)
AlexNet	224×224	46.8M	938M	0.03	91%
VGG, quarter width	224×224	43.1M	512M	0.1	93%
VGG, quarter width	224×224	43.1M	512M	0.03	92%
VGG, quarter width, pooled head	224×224	0.6M	470M	0.03	89–90%
NiN	224×224	3.4M	798M	0.03	89%
GoogLeNet	96×96	6.0M	276M	0.03	91%

How much can we read from this table? Less than one would like. Ten epochs on upsampled 28×28 images are a small test, and the networks end within about four points of each other. The peak rate alone moves VGG by a point and a half, so differences of a point between networks trained at different peaks mean little. Three observations nevertheless hold across repeated runs.

First, arithmetic alone does not decide accuracy. The narrow VGG is the most accurate network at either peak with about half of AlexNet's multiply-adds, while NiN performs almost as many as AlexNet and ends about two points below it. Second, GoogLeNet matches AlexNet with an eighth of its parameters and from images with less than a fifth of the pixels. Its small count of multiply-adds, however, is due to those smaller images and not to the design: the count grows roughly in proportion to the number of pixels, so at 224×224 GoogLeNet would perform about 1.5 billion multiply-adds, more than AlexNet. Third, on this benchmark the dense head does contribute. On the same VGG body at the same peak, global average pooling with one linear layer saves 99% of the parameters and loses about two points. A network without a dense head can still match AlexNet, as GoogLeNet does. On ImageNet the evidence is clearer: in the challenge of 2014, GoogLeNet won with about 7 million parameters per network and a lower error than VGG, whose networks had up to 144 million.

GoogLeNet's hand-tuned branches have aged least well and are rare in current networks. What survives of them is more regular: grouped convolutions, as in ResNeXt (Section 7.4.5.1), are multi-branch blocks whose branches are all alike.

None of these networks normalizes its activations, and none is much deeper than twenty layers. VGG's authors trained their deepest networks by first training shallower ones and copying their weights, and GoogLeNet added auxiliary classifiers in the middle of the network to keep its gradients from vanishing; our implementations omit both. We met a third symptom ourselves: the peak learning rate had to be chosen network by network, close to an edge beyond which training fails. All three point to the same missing piece, a way to keep the scale of the signals under control at every layer.

7.2.5 Summary

After AlexNet, networks were built from blocks rather than individual layers. VGG repeats a block of 3×3 convolutions and pooling, so that a network is a list of block sizes and depth comes from stacking small kernels. NiN mixes channels with 1×1 convolutions and replaces the fully connected head, which holds over 90% of the parameters of AlexNet and VGG, by global average pooling with no parameters. GoogLeNet runs kernels of several sizes in parallel, makes them affordable with 1×1 bottlenecks, and organizes the network into stem, body, and head. Trained on Fashion-MNIST, each at the highest peak learning rate at which it trains reliably, the narrow VGG is the most accurate of the four, and GoogLeNet matches AlexNet with an eighth of its parameters. On the same VGG body, replacing the dense head by global average pooling saves 99% of the parameters and loses about two points.

7.2.6 Exercises

1. code **Translations and the head.** Train `VGG` and `PooledVGG` as in this section, both at a peak of 0.03, and keep the two trained networks.
    1. Shift every resized 224×224 validation image horizontally by 32 pixels, filling with zeros, and measure how much the validation accuracy of each network drops.
    2. Relate the result to the argument about translations in this section. Why is the pooled network not exactly invariant to the shift?
2. code **The 1×1 convolutions of NiN.** Remove both 1×1 convolutions from every `nin_block`, so that each block is a single convolution followed by a ReLU. In the last block the 3×3 convolution must then emit the ten class scores itself, without a ReLU. Train the network as `NiN`.
    1. How many parameters and multiply-adds does the network lose?
    2. Compare the validation accuracy with that of `NiN`. Which of the four blocks needs its 1×1 convolutions most? Test your answer by restoring them in one block at a time.
3. **VGG-16 from its configuration.** VGG-16 has 13 convolutional layers in five blocks with `arch=((2, 64), (2, 128), (3, 256), (3, 512), (3, 512))`, and three fully connected layers of 4096, 4096, and 1000 units, on 224×224 color images.
    1. Count its parameters and multiply-adds by hand, separately for the convolutional

and the fully connected layers, and check your total against the 138 million parameters reported by Simonyan and Zisserman (2015) .
2. Which layer has the most parameters? Which layers perform the most multiply-adds, and why do several of them tie?

4. code **Native resolution.** Fashion-MNIST images are 28×28, and the training runs of this section upsample them eightfold to 224×224, which spends computation without adding information.
   1. Train `VGG` as in this section, with `arch=((1, 16), (1, 32), (2, 64), (2, 128), (2, 128))` and `lr=0.1`, on images resized to 56×56 and to 84×84. The input size of the first fully connected layer changes with the resolution. Compare validation accuracy and time per epoch with the same model trained at 224×224, and give the spatial size of the feature map that enters the fully connected layers at each of the three resolutions.
   2. `GoogLeNet` accepts a 28×28 input as written, because its convolutions and pooling layers pad so that every stride-2 step halves the resolution, rounding up. Compute the resolution at which each of the three groups of Inception blocks operates on a 28×28 input. What do the 3×3 and 5×5 branches compute at the smallest of these resolutions?

5. **Where the savings come from.** ☒ Use the counts of this section and the table of AlexNet's layers, Table 7.1.
   1. For VGG-11, `NiN`, and `GoogLeNet` on a single-channel 224×224 image with ten classes, split the parameters and the multiply-adds between the convolutional layers and the classification head.
   2. Relative to VGG-11, does `NiN` save more parameters in its body or in its head? And `GoogLeNet`? Answer the same questions for the multiply-adds, and name the design choice responsible for each saving.

   Adapted from Simon Prince, Understanding Deep Learning, Problem 10.18.

6. code **Single-branch ablation.** ☒ Build a variant of `GoogLeNet` whose Inception blocks keep only the 3×3 branch with its 1×1 reduction. Let the 3×3 convolution emit as many channels as the original block, and choose the width of the 1×1 reduction so that each block's parameter count stays within 10% of the original. Train both networks as `GoogLeNet` is trained in this section and compare validation accuracy. What does the result say about the value of mixing kernel sizes within one block?

7.3 Normalization Layers

When we trained LeNet in Section 6.6, the learning rate that suited the MLP did not work. At a learning rate of 0.1 its validation accuracy stayed at chance for the first few epochs and reached only about 73% after ten; we had to raise the rate to 2.0, twenty times larger, and train for 30 epochs to reach about 89%. A network of five layers should not need such care. Nor should the networks of Section 7.2, where a peak rate of 0.1 suited one network and made others diverge or stall. Why does the right learning rate depend so strongly on the network, and can a layer inside the network remove that dependence?

The cause is the scale of the signals inside the network. Each of LeNet's four sigmoids multiplies the gradient by at most $1/4$ on the way back, so the first layers receive much smaller gradients than the last, and no single learning rate suits all of them. Depth makes matters

worse, since every layer changes the scale of the signals that enter the next one. Have the ReLU and a careful initialization (Section 4.4.2.2) not solved this already? Only in part: they set the scales right at the start of training, and nothing holds them there once the weights move.

Batch normalization (Ioffe and Szegedy 2015) controls these scales inside the network, throughout training. Before the nonlinearity of a layer, it standardizes each feature with the mean and the variance of the current minibatch, then applies a learned scale and shift. This rescales the signal on the way forward and its gradient on the way back, and it makes the layer indifferent to the length of its weights. We will see that LeNet with batch normalization trains well at the MLP's learning rate, and at rates a hundred times larger. Together with the residual connections of the next section, batch normalization made networks with more than a hundred layers trainable, although why it helps as much as it does is still argued over. Its statistics, however, couple the examples of a minibatch, which fails when minibatches are small. The last part of this section introduces layer and group normalization, which compute their statistics within each example.

7.3.1 Batch Normalization

7.3.1.1 Standardizing over a Minibatch

Consider one feature computed by a fully connected layer, say the input to the nonlinearity of one hidden unit, and a minibatch $\mathcal{B}$ of m examples. The layer produces one value x_i of this feature for each example i. Batch normalization computes the mean and the variance of these m values,

$$\hat{\mu}_{\mathcal{B}} = \frac{1}{m}\sum_{i=1}^{m} x_i \quad \text{and} \quad \hat{\sigma}^2_{\mathcal{B}} = \frac{1}{m}\sum_{i=1}^{m}(x_i - \hat{\mu}_{\mathcal{B}})^2, \tag{7.2}$$

and replaces each value by

$$\mathrm{BN}(x_i) = \gamma\,\frac{x_i - \hat{\mu}_{\mathcal{B}}}{\sqrt{\hat{\sigma}^2_{\mathcal{B}} + \epsilon}} + \beta. \tag{7.3}$$

The fraction standardizes the feature: over the minibatch it has mean 0 and variance $\hat{\sigma}^2_{\mathcal{B}}/(\hat{\sigma}^2_{\mathcal{B}} + \epsilon)$, just below 1. The small constant $\epsilon > 0$, typically 10^{-5}, prevents a division by zero when the feature takes the same value on every example of the minibatch. A layer with several features normalizes each of them separately, with its own mean and variance.

For example, the four values $(2, 4, 6, 8)$ have mean 5 and variance $(9 + 1 + 1 + 9)/4 = 5$. Ignoring ϵ, they are standardized to $(x_i - 5)/\sqrt{5} \approx (-1.34, -0.45, 0.45, 1.34)$. The values $(20, 40, 60, 80)$ and $(102, 104, 106, 108)$ are standardized to the same four numbers: the standardization removes the scale and the offset of the feature.

Mean 0 and variance 1 need not be what the next layer works best with. A sigmoid, for instance, is nearly linear on inputs of unit variance. The *scale* γ and the *shift* β in Equation 7.3 restore this freedom. They are parameters learned with the weights, one pair per feature, initialized to $\gamma = 1$ and $\beta = 0$. Nothing is lost by standardizing first: with γ and β equal to the standard deviation and the mean of the feature over the data, the layer would restore

the original scale and offset, up to the fluctuation of the minibatch statistics. Batch normalization therefore does not restrict what a network can represent.

7.3.1.2 Scale Invariance

Removing the scale of a feature has two consequences. The first is simple, and it is the one that matters for LeNet. A feature that barely varies over the minibatch, such as one computed from sigmoid outputs that all lie near $1/2$, is divided by its small $\widehat{\sigma}_{\mathcal{B}}$ on the way forward, and by the chain rule its gradient is multiplied by the same factor $1/\widehat{\sigma}_{\mathcal{B}}$ on the way back. A signal that has become too small is enlarged, and its gradient with it.

The second consequence is subtler and concerns the weights. Let $x_i = \mathbf{w}^\top \mathbf{h}_i + b$ be computed from the inputs $\mathbf{h}_i$ of example i. Multiplying $\mathbf{w}$ and b by a constant $a > 0$ multiplies every x_i, their mean, and their standard deviation by a, so, ignoring ϵ,

$$\mathrm{BN}(a\mathbf{w}^\top \mathbf{h}_i + ab) = \mathrm{BN}(\mathbf{w}^\top \mathbf{h}_i + b).$$

The loss therefore depends on the direction of the weights but not on their length. Differentiating both sides with respect to $\mathbf{w}$ shows that the gradient at $a\mathbf{w}$ is $1/a$ times the gradient at $\mathbf{w}$ (Ioffe and Szegedy 2015): small weights receive large gradients and large weights small ones.

What happens when the learning rate is too large? Since the loss does not change along $\mathbf{w}$, the gradient $\mathbf{g}$ is orthogonal to $\mathbf{w}$, and a step of gradient descent with learning rate η makes the weights longer: $\|\mathbf{w} - \eta\mathbf{g}\|^2 = \|\mathbf{w}\|^2 + \eta^2\|\mathbf{g}\|^2$. A step that is too large makes them much longer, which shrinks all later gradients of the layer. An overlarge rate thus corrects itself to some degree (Arora et al. 2019).

The argument has conditions, and they are worth knowing. It describes plain gradient descent and SGD without weight decay; with weight decay the length of the weights does not keep growing but settles where growth and decay balance. It covers only the weights that feed a normalization, since only they are scale-invariant, and the output layer of a classifier is not among them. And it promises nothing about the first steps, in which the loss can still blow up before the weights have grown. Within these limits it is the best argument we have for why normalized networks tolerate large learning rates.

7.3.1.3 Fully Connected and Convolutional Layers

In a fully connected layer with weights $\mathbf{W}$, bias $\mathbf{b}$, and activation function ϕ, batch normalization goes between the affine map and the nonlinearity, as in the original paper (Ioffe and Szegedy 2015):

$$\mathbf{h} = \phi(\mathrm{BN}(\mathbf{W}\mathbf{x} + \mathbf{b})). \tag{7.4}$$

Each of the outputs of the affine map is one feature with its own statistics and its own γ_j and β_j.

A convolutional layer applies the same kernel at every position, and batch normalization follows suit: it treats each output *channel* as one feature and computes its mean and variance over the examples of the minibatch *and* over all positions. With m examples and an output of height p and width q, each channel's statistics average $m \cdot p \cdot q$ numbers. For the first

convolutional layer of LeNet, which produces six channels of 28×28, a minibatch of 128 images gives $128 \cdot 28 \cdot 28 = 100{,}352$ numbers per channel, against 128 per feature in a fully connected layer. Each channel has a single γ and β, so the normalized layer still applies the same function at every position.

7.3.1.4 Training and Prediction

At prediction time a model may see one image at a time, and its output for an image should not depend on which other images happen to share its minibatch. Batch normalization therefore uses two different statistics in two modes. In *training mode* it normalizes with the statistics of the current minibatch, as in Equation 7.3. It also keeps running estimates $\hat{\mu}$ and $\hat{\sigma}^2$ of the mean and the variance, starting from 0 and 1, and updates them after every minibatch,

$$\hat{\mu} \leftarrow \rho\,\hat{\mu} + (1-\rho)\,\hat{\mu}_{\mathcal{B}} \quad \text{and} \quad \hat{\sigma}^2 \leftarrow \rho\,\hat{\sigma}^2 + (1-\rho)\,\hat{\sigma}^2_{\mathcal{B}}. \tag{7.5}$$

The *decay* ρ sets the memory of these exponential moving averages: the minibatch k steps back contributes with weight $(1-\rho)\rho^k$. With $\rho = 0.9$ these weights halve about every seven steps, and the estimates average over roughly the last $1/(1-\rho) = 10$ minibatches, a small fraction of the 469 minibatches of 128 images in one epoch of Fashion-MNIST. In *prediction mode* the layer normalizes with the running estimates and leaves them unchanged, so it computes a fixed affine function of each feature. Like dropout (Section 4.6), the layer thus behaves differently in training and at prediction.

Unfortunately, some framework layers call ρ the *momentum* of the layer and others $1-\rho$, and the name has nothing to do with the momentum method of optimization. Throughout this chapter we use $\rho = 0.9$ and $\epsilon = 10^{-5}$ and pass them explicitly where the layer's defaults differ.

7.3.2 Implementation from Scratch

The function `batch_norm` implements Equation 7.3 and Equation 7.5 for both modes and for both kinds of layer.

Images are stored as (examples, channels, height, width), so the channel axis is 1.

```
def batch_norm(X, gamma, beta, moving_mean, moving_var, eps, momentum,
               training):
    if not training:
        # Prediction mode: normalize with the running estimates
        X_hat = (X - moving_mean) / torch.sqrt(moving_var + eps)
    else:
        assert len(X.shape) in (2, 4)
        # Fully connected: average over examples. Convolutional: over
        # examples and positions, one mean and variance per channel
        dims = 0 if len(X.shape) == 2 else (0, 2, 3)
        mean = X.mean(dim=dims, keepdim=True)
        var = ((X - mean) ** 2).mean(dim=dims, keepdim=True)
        X_hat = (X - mean) / torch.sqrt(var + eps)
        # Update the running estimates; `momentum` is the decay rho
        moving_mean = momentum * moving_mean + (1.0 - momentum) * mean
```

```
            moving_var = momentum * moving_var + (1.0 - momentum) * var
        Y = gamma * X_hat + beta  # Scale and shift
        return Y, moving_mean.detach(), moving_var.detach()
```

The layer `BatchNorm` holds two kinds of state. The scale `gamma` and the shift `beta` are parameters, updated by the optimizer. The running estimates `moving_mean` and `moving_var` are not parameters: the optimizer must leave them alone, but they belong to the model, which saves them and moves them to the GPU with its parameters.

```
class BatchNorm(nn.Module):
    # `num_features`: outputs of a fully connected layer or channels of a
    # convolutional layer. `num_dims`: 2 or 4, the number of input axes
    def __init__(self, num_features, num_dims):
        super().__init__()
        shape = (1, num_features) if num_dims == 2 else (1, num_features, 1, 1)
        # Scale and shift, learned, start at 1 and 0
        self.gamma = nn.Parameter(torch.ones(shape))
        self.beta = nn.Parameter(torch.zeros(shape))
        # Running estimates, not learned, start at 0 and 1
        self.register_buffer('moving_mean', torch.zeros(shape))
        self.register_buffer('moving_var', torch.ones(shape))

    def forward(self, X):
        Y, moving_mean, moving_var = batch_norm(
            X, self.gamma, self.beta, self.moving_mean, self.moving_var,
            eps=1e-5, momentum=0.9, training=self.training)
        self.moving_mean.copy_(moving_mean)
        self.moving_var.copy_(moving_var)
        return Y
```

`register_buffer` registers a tensor that `model.to(device)` moves and `state_dict()` saves, but that `parameters()` does not return. The attribute `self.training` is set by `model.train()` and `model.eval()`, which the Trainer calls before training and validation.

To check the implementation, we apply our layer and the framework's, both with $\rho = 0.9$ and $\epsilon = 10^{-5}$, to the same minibatch of 128 six-channel 28×28 feature maps whose entries have mean 2 and standard deviation 3. The cell reports the largest per-channel mean of our output, which should be 0; its average per-channel variance, which should be 1; its largest difference from the framework's output; and our running means after this one step.

```
X = 2 + 3 * torch.randn(128, 6, 28, 28)
bn, bn_ref = BatchNorm(6, num_dims=4), nn.BatchNorm2d(6)  # rho 0.9, eps 1e-5
with torch.no_grad():
    Y, Y_ref = bn(X), bn_ref(X)  # New layers are in training mode
{'largest channel mean': float(Y.mean(dim=(0, 2, 3)).abs().max()),
 'average channel variance': float(Y.var(dim=(0, 2, 3),
                                         unbiased=False).mean()),
 'largest difference': float((Y - Y_ref).abs().max()),
 'running means': bn.moving_mean.reshape(-1)}
```

```
{'largest channel mean': 1.1894319840166645e-07,
 'average channel variance': 0.9999990463256836,
```

```
'largest difference': 9.5367431640625e-07,
'running means': tensor([0.2005, 0.1997, 0.2001, 0.2010, 0.1976, 0.2012])}
```

Every channel of the output has mean 0 and variance 1 up to single-precision roundoff, and our layer agrees with the framework's to within about 10^{-6}. After one step the running means have moved $1-\rho$, a tenth of the way, from their starting value 0 toward the minibatch means of about 2.

`nn.BatchNorm2d(6)` uses PyTorch's default `momentum=0.1`, which is our $\rho = 0.9$.

7.3.3 LeNet with Batch Normalization

Let us return to the network that started this section. We add batch normalization to the LeNet of Section 6.6, with the same Xavier initialization, and use the framework layers, which we have just seen compute what ours does. LeNet has four hidden layers: two convolutional layers with 6 and 16 channels and two fully connected layers with 120 and 84 units. `BNLeNet` inserts a batch normalization layer after each of them, before its sigmoid, and drops their biases: batch normalization subtracts the mean of each feature, which removes any bias added before it, and its shift β takes over the role of the bias. Each of the $6 + 16 + 120 + 84 = 226$ normalized features thus trades one bias for a scale and a shift, 226 parameters on top of LeNet's 61,706.

```
class BNLeNet(d2l.Classifier):
    """LeNet with batch normalization before each hidden sigmoid."""
    def __init__(self, lr=0.1, num_steps=4690, num_classes=10):
        super().__init__()
        self.save_hyperparameters()
        self.net = nn.Sequential(  # No biases before batch normalization
            nn.LazyConv2d(6, kernel_size=5, padding=2, bias=False),
            nn.LazyBatchNorm2d(), nn.Sigmoid(),
            nn.AvgPool2d(kernel_size=2, stride=2),
            nn.LazyConv2d(16, kernel_size=5, bias=False),
            nn.LazyBatchNorm2d(), nn.Sigmoid(),
            nn.AvgPool2d(kernel_size=2, stride=2),
            nn.Flatten(),
            nn.LazyLinear(120, bias=False), nn.LazyBatchNorm1d(), nn.Sigmoid(),
            nn.LazyLinear(84, bias=False), nn.LazyBatchNorm1d(), nn.Sigmoid(),
            nn.LazyLinear(num_classes))

    weight_init = d2l.LeNet.weight_init  # Xavier weights, zero biases
```

`LazyBatchNorm2d` normalizes each channel of a four-axis input and `LazyBatchNorm1d` each feature of a two-axis one. Their defaults, `momentum=0.1` and `eps=1e-5`, are our $\rho = 0.9$ and ϵ. `BNLeNet` reuses LeNet's `weight_init`; only the output layer has a bias for it to set to zero.

Prediction mode raises one more issue. The running averages summarize the statistics of the last ten or so minibatches, computed with weights that have changed since. While the weights still move quickly, the running averages describe a slightly different network from the one we evaluate. This is no mere technicality: with a constant learning rate of 0.1, the prediction-mode validation accuracy of `BNLeNet` drops by ten points or more in single epochs of some runs, the last epoch included. We therefore let the learning rate decay to zero during training. As the rate shrinks, the weights stop moving and the running averages catch up with them. To this end we reuse the schedule Equation 7.1 with which we trained

AlexNet (Section 7.1), a linear warmup followed by a half-cosine decay to zero; Chapter 9 treats such schedules. The warmup takes one tenth of the `num_steps` steps of training. For a fair comparison, LeNet gets the same schedule through a subclass.

```
@d2l.add_to_class(BNLeNet)
def configure_optimizers(self):
    optim = torch.optim.SGD(self.parameters(), lr=self.lr)
    schedule = torch.optim.lr_scheduler.LambdaLR(optim, lambda t: (
        d2l.cosine_warmup(t, self.num_steps, self.num_steps // 10)))
    return optim, schedule  # The Trainer steps it after every minibatch

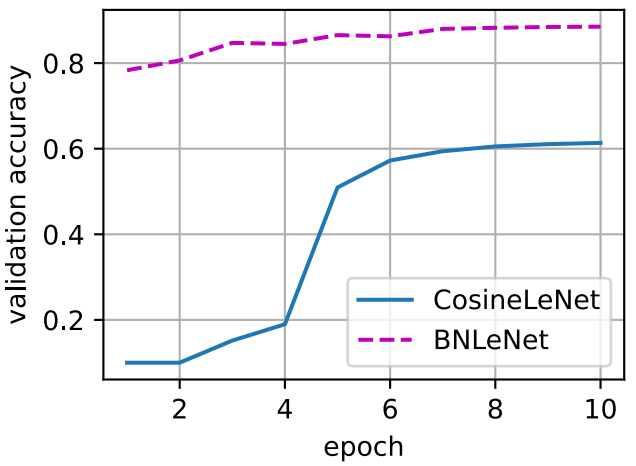
class CosineLeNet(d2l.LeNet):
    def __init__(self, lr=0.1, num_steps=4690, num_classes=10):
        super().__init__(lr, num_classes)
        self.num_steps = num_steps

    configure_optimizers = BNLeNet.configure_optimizers
```

We train LeNet and `BNLeNet` with identical settings, a peak learning rate of 0.1 (the MLP's), minibatches of 128 images, and 10 epochs, and keep the validation accuracy of each after every epoch.

```
data = d2l.FashionMNIST(batch_size=128)
num_steps = 10 * len(data.train_dataloader())
curves = {}
for model in (CosineLeNet(0.1, num_steps), BNLeNet(0.1, num_steps)):
    trainer = d2l.Trainer(max_epochs=10, num_gpus=1)
    trainer.board.display = False  # We plot the accuracies ourselves
    trainer.fit(model, data)
    curves[type(model).__name__] = trainer.history['val_acc']
```

```
d2l.plot(list(range(1, 11)), list(curves.values()), 'epoch',
         'validation accuracy', legend=list(curves))
```



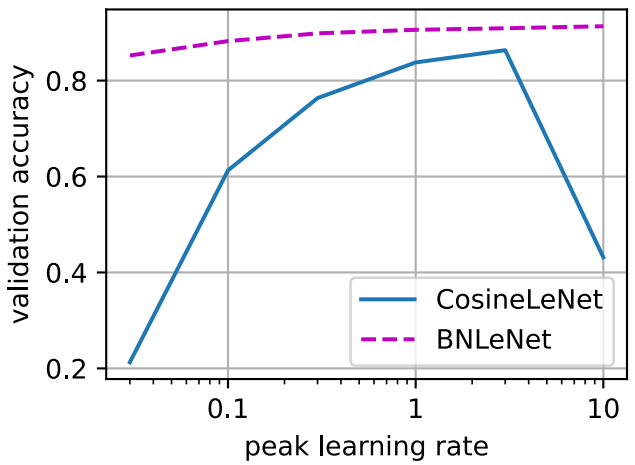
```
{name: round(acc[-1], 3) for name, acc in curves.items()}  # After 10 epochs
```

```
{'CosineLeNet': 0.614, 'BNLeNet': 0.885}
```

LeNet sits at chance, 10%, for the first two to five epochs, and after ten it reaches about 60%, less in runs that stay at chance longer. That it falls short of the 73% of Section 6.6 is plausible, since the schedule halves the average learning rate; we have not tested this explanation. With batch normalization the same network is near 70% or above after the first epoch and reaches about 88% after ten, close to the 88–90% that LeNet reaches with twenty times the learning rate and three times the epochs. Its curve is flat over the last epochs, where the learning rate has decayed. Batch normalization added 226 parameters and a few operations per activation. It changed the scale of every layer's signal and gradient, and left the capacity of the network as it was.

Both networks ran at a single learning rate, and 0.1 suits LeNet badly. Does batch normalization merely move the best learning rate, or does it make the choice matter less? We train both networks again with peak rates from 0.03 to 10, a factor of 300, and plot the validation accuracy after 10 epochs against the peak rate.

```
lrs = (0.03, 0.1, 0.3, 1, 3, 10)
final = {'CosineLeNet': [], 'BNLeNet': []}
for lr in lrs:
    for cls in (CosineLeNet, BNLeNet):
        trainer = d2l.Trainer(max_epochs=10, num_gpus=1)
        trainer.board.display = False
        trainer.fit(cls(lr, num_steps), data)
        final[cls.__name__].append(trainer.history['val_acc'][-1])
d2l.plot(list(lrs), list(final.values()), 'peak learning rate',
         'validation accuracy', legend=list(final), xscale='log')
```



Without batch normalization LeNet does well only at rates near 1 to 3, where it reaches 84–87%. Below that band it learns slowly or not at all in 10 epochs, and at 10 the outcome is erratic: some runs reach the mid-80s, others never leave chance. With batch normalization every rate in the range, from 0.03 to 10, gives about 85% or more, and rates of 1 and above

reach about 91%, three points more than the 0.1 we used above. The plain network has to be tuned to within a factor of three. The normalized one works over a factor of 300, and this is the answer to the question that opened the section. It is also the tolerance of large rates that scale invariance led us to expect, although we should not expect it to continue indefinitely: the output layer is not normalized.

Why batch normalization helps so much is less settled than what it computes. Its authors attributed the effect to a reduction of *internal covariate shift*, the change in the distribution of a layer's inputs as the layers before it learn (Ioffe and Szegedy 2015). Later measurements found that batch normalization does not reduce this shift, and that it makes the loss and its gradient change more smoothly with the parameters instead (Santurkar et al. 2018). What can be said with confidence is narrower. Batch normalization rescales the signal and its gradient at every layer, and it makes each layer invariant to the scale of its weights. As the sweep showed, normalized networks tolerate larger learning rates. The noise of the minibatch statistics is also widely believed to act as a regularizer (Luo et al. 2018; Brock et al. 2021); we may think of it as a relative of the noise of dropout. Why these effects add up to faster training of very deep networks is still argued over. The method spread on the strength of its results, well ahead of its explanation (Lipton and Steinhardt 2018).

7.3.4 Normalization without Batch Statistics

The output of batch normalization for one example depends on the other examples in its minibatch, a strange property for a layer to have. This coupling causes two problems. First, the statistics are only as good as the minibatch is large. In a fully connected layer, a minibatch of one example makes every standardized feature 0, since each value equals its own mean; the layer then outputs β whatever the input. Object detection and semantic segmentation train on large images, so memory often limits them to one or two images per GPU, where the statistics are noisiest. Second, the network computes different functions in training and in prediction mode. When the running estimates describe the data at prediction time poorly, for instance after fine-tuning on data from another domain, accuracy drops in a way that is hard to trace to a normalization layer.

Both problems disappear when each example is normalized with statistics of its own. Figure 7.6 shows the options for a convolutional layer, whose activations have an example axis N, a channel axis C, and the spatial positions $H \cdot W$. They differ only in the set of numbers that share one mean and one variance.

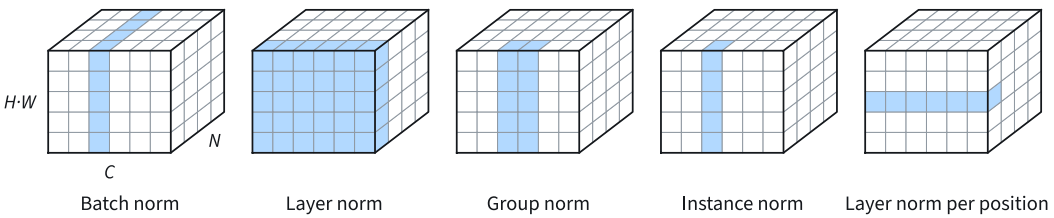


Figure 7.6: Normalization methods for activations with examples N, channels C, and spatial positions $H \cdot W$. The shaded numbers share one mean and one variance. Batch normalization averages each channel over the minibatch and all positions; layer normalization averages each example over all channels and positions; group normalization over a group of channels and all positions; instance normalization over a single channel. The layer normalization of transformers and ConvNeXt averages over the channels at one position.

7.3.4.1 Layer Normalization

Layer normalization (Ba et al. 2016) standardizes the features of each example separately. For the n features $\mathbf{x} = (x_1, \dots, x_n)$ of one example it computes

$$\mathrm{LN}(\mathbf{x})_j = \gamma_j \frac{x_j - \hat{\mu}}{\sqrt{\hat{\sigma}^2 + \epsilon}} + \beta_j \quad \text{with} \quad \hat{\mu} = \frac{1}{n}\sum_{j=1}^{n} x_j \quad \text{and} \quad \hat{\sigma}^2 = \frac{1}{n}\sum_{j=1}^{n}(x_j - \hat{\mu})^2.$$

This is Equation 7.3 with the roles of examples and features exchanged: the average runs over the features of one example instead of over the examples of one feature, while the scale γ_j and the shift β_j remain per feature. No other example enters, so layer normalization computes the same function at every minibatch size and in both modes, and it keeps no running estimates. Which features are averaged together depends on the model, and the name is used rather loosely. For a convolutional layer it can mean all channels and positions of an example, the second panel of Figure 7.6. Transformers normalize the vector of each token, and the convolutional network ConvNeXt (Section 7.7) likewise normalizes the channel vector at each position, the last panel, without pooling over positions.

7.3.4.2 Group and Instance Normalization

For convolutional layers, *group normalization* (Wu and He 2018) divides the c channels into G groups of c/G channels and standardizes each example separately over each group, that is, over $(c/G) \cdot p \cdot q$ numbers, followed by a scale and a shift per channel. With $G = 1$ it normalizes each example over all channels and positions at once, the layer normalization of Figure 7.6. With $G = c$ it normalizes every channel of every example separately; this special case is called *instance normalization*. Intermediate group counts, typically $G = 32$, give each group enough numbers for stable statistics without forcing unrelated channels to share them. We check on a minibatch of 32 examples with 16 channels and 4 groups that every (example, group) pair is standardized, and that normalizing the first example alone gives the same output as normalizing it within the minibatch.

```
X = torch.randn(32, 16, 8, 8)
gn = nn.GroupNorm(4, 16).requires_grad_(False)  # 4 groups of 4 channels
Y = gn(X)
G = Y.reshape(32, 4, -1)  # Collect each (example, group) pair's numbers
(G.mean(dim=2).abs().max(), G.var(dim=2, unbiased=False).mean(),
 (gn(X[:1]) - Y[:1]).abs().max())
```

```
(tensor(2.3283e-08), tensor(1.0000), tensor(0.))
```

Each group has mean 0 and variance 1 up to roundoff, and the first example is normalized the same way alone as within the minibatch. Does this independence cost accuracy? With large minibatches, a little: for a ResNet-50 trained on ImageNet with 32 images per GPU, Wu and He (2018) report a top-1 error of 23.6% with batch normalization and 24.1% with group normalization. With 2 images per GPU the error of batch normalization rises to 34.7%, while that of group normalization stays at 24.1%. Group normalization is therefore common where minibatches are small, as in detection and segmentation.

7.3.4.3 Normalizer-Free Networks

Normalization layers are not the only way to keep the scales of a deep network under control. Normalizer-free networks (NFNets) (Brock et al. 2021) train ResNet-style networks without any normalization layer, to an ImageNet accuracy that was the best reported at the time for a model trained on ImageNet alone. They standardize the weights of each convolution instead of its outputs, which keeps throughout training the scale that a careful initialization (Section 4.4.2.2) sets only at the start, and they clip a gradient whose norm is large relative to the norm of the weights it updates. We will not use these networks. They matter here as evidence: a network that trains this well once the scale of its weights is held fixed by other means suggests that controlling scale is the main thing a normalization layer does.

7.3.5 Summary

Batch normalization standardizes each feature of a layer, or each channel of a convolutional layer, with the mean and the variance of the current minibatch, then applies a learned scale and shift. This rescales the signal and its gradient at every layer and makes the layer invariant to the scale of its weights. LeNet, whose sigmoids leave the first layers with small gradients, then trains at every peak learning rate from 0.03 to 10, where the plain network needs one between about 1 and 3. Why the method helps very deep networks as much as it does is not settled. At prediction time the layer uses running averages of the statistics instead. Its dependence on the minibatch fails when minibatches are small; layer and group normalization compute their statistics within each example and avoid both the coupling and the two modes. The next section combines batch normalization with residual connections to train much deeper networks.

7.3.6 Exercises

1. **Bias before batch norm.** `BNLeNet` builds its four normalized layers without a bias. Show from Equation 7.3 that adding a bias $\mathbf{b}$ before the normalization changes neither the output in training mode nor the gradient with respect to the weights, and that the gradient with respect to $\mathbf{b}$ is zero. What happens to such a bias in prediction mode?
2. code **Learning-rate ceiling.** In the sweep of this section `BNLeNet` trained at every peak rate up to 10.
    1. Extend the sweep for `BNLeNet` to peak rates of 30, 100, and 300. Where does training fail?
    2. The output layer is the only layer with weights that no normalization follows. Test whether it sets the ceiling: give it a learning rate of its own, fixed at 0.1, and repeat the first part.
    3. Replace the sigmoids of both networks by ReLUs, with He initialization, and repeat the sweep of this section. How does the band of good rates of the plain network change, and why?
3. code **Leave-one-out ablation.** Does every layer need batch normalization? `BNLeNet` normalizes the output of each of its four hidden layers. Remove batch normalization from one layer at a time and retrain. Report, as a table, the validation-accuracy cost of each removal, and identify the layer whose removal hurts most.
4. code **Running averages and recomputed statistics.** The decay ρ in Equation 7.5 sets how many minibatches the running estimates remember.
    1. After how many steps does the weight of a minibatch in the running mean fall

below 1% of the weight of the newest minibatch, for $\rho = 0.9$ and for $\rho = 0.99$? Compare both with the 469 steps of an epoch.
   2. Train `BNLeNet` for 10 epochs with a constant learning rate of 0.1, the `configure_optimizers` of `d2l.Classifier`, instead of the cosine schedule. After every epoch, evaluate a copy of the model twice: once in prediction mode as it is, and once after recomputing its statistics for the current weights. To recompute them, reset the running estimates and pass 100 training minibatches through the copy in training mode, without updating the weights, averaging the minibatch statistics with equal weights. Plot both validation curves, add the curve of this section's cosine schedule, and explain the differences.
   3. Repeat the second part with $\rho = 0.99$. Does a longer memory remove the difference? Why or why not?
5. code **Minibatch coupling.** The output of batch normalization for one example depends on the other examples in its minibatch. Train `BNLeNet` as in this section. Measure its validation accuracy in prediction mode, then in training mode with validation batches of 128 images, and then in training mode with validation batches of one image. Measure in prediction mode first, since every training-mode forward pass updates the running estimates. Explain the three accuracies. If the layer refuses a training-mode minibatch of one example, explain why, and measure with minibatches of two instead.
6. code **Small-batch training.** ▯ Build a variant of `BNLeNet` that uses group normalization with two groups after each convolutional layer and layer normalization (Section 7.3.4.1) after each hidden fully connected layer, in place of batch normalization. Train `BNLeNet` and the variant for two epochs at batch size 4 and again at batch size 128, keeping the other hyperparameters of this section. Report the validation accuracy of all four runs. How does the accuracy of each network change when the batch size drops from 128 to 4, and why?
7. code **Backward pass of batch norm.** ▯ The batch statistics in Equation 7.3 depend on every element of the minibatch, so the backward pass is not elementwise.
   1. Derive, by hand, the gradient of a scalar loss with respect to each input of the batch-norm layer, working through the forward steps: mean, centering, variance, normalization, and the affine scale and shift.
   2. Implement your expression and verify it against the gradient computed by automatic differentiation.
   3. Show from your expression that the gradient with respect to the inputs sums to zero over the minibatch, and explain why.

   Adapted from Simon Prince, Understanding Deep Learning, Problem 11.5.
8. [extended] **Beyond mean and variance.** ▯▯ Batch normalization standardizes each channel separately. For each of the following alternatives, describe how it would be applied at a layer and what it costs in computation, parameters, and estimation variance. Implement at least one of them in place of our `BatchNorm` in a copy of `BNLeNet` and compare its validation accuracy with that of batch normalization.
   1. The probability integral transform, which maps each channel through its empirical cumulative distribution function.
   2. Whitening with a full covariance estimate across the c channels.
   3. Whitening with a structured covariance estimate, for example a block-diagonal or a low-rank-plus-diagonal matrix.

7.4 Residual Networks

On ImageNet, depth and accuracy had so far grown together, from the eight layers of AlexNet to the 22 of GoogLeNet, and the batch normalization of Section 7.3 made such depth easier to train. We might expect that adding still more layers keeps helping, at worst until the network starts to overfit. He et al. (2016) tested this expectation. They trained *plain* networks, stacks of convolutions with batch normalization and ReLU in the style of VGG, at several depths. On CIFAR-10 a plain network with 56 layers ended training with a higher *training* error than one with 20 layers. A deeper network that fits even its training data worse does not overfit; the optimizer fails to find a good setting of its parameters. This *degradation problem* is puzzling, because the deeper network could copy the shallower one and pass its result through the extra layers unchanged, so a setting that is at least as good exists. Why does training not find it, and how can we change the network so that it does?

Residual networks (ResNets) make that setting easy to reach. Each block adds a learned correction to its input instead of replacing it, so a block whose correction is zero passes its input through unchanged. This small change was enough. On ImageNet, where a plain network with 34 layers trained worse than one with 18, the residual network with 34 layers was almost three points more accurate than its 18-layer relative, and residual networks with 152 layers won the ImageNet challenge of 2015.

We first ask what the extra layers of a deeper network would have to do in order not to hurt, and build the residual block. Then we repeat the experiment ourselves: on Fashion-MNIST we train plain and residual networks of 20 and 56 layers and watch the degradation appear and disappear. After that we build and train the 18-layer ResNet. Two variants close the section: ResNeXt makes the block wider at the same cost, and DenseNet concatenates where ResNet adds.

7.4.1 Function Classes and Depth

It is instructive to describe a network by the set of functions it can represent as its parameters vary, its *function class* $\mathcal{F}$. Training searches $\mathcal{F}$ for the function that best fits the training data. When does a larger architecture, with class $\mathcal{F}'$, have a better function to offer? Certainly if the classes are *nested*, $\mathcal{F} \subseteq \mathcal{F}'$: the larger class still contains the best function of the smaller one, so the smallest training loss available in it can only be lower. For classes that are not nested there is no such guarantee, since the larger class may have lost the function that worked. Figure 7.7 illustrates the difference, with a star for the function we would like to find. Note, however, that nesting is a statement about the best function in each class. It does not promise that the optimizer finds that function, nor that the function generalizes better.

Adding a layer to a network nests the function classes if the new layer can compute the identity $f(\mathbf{x}) = \mathbf{x}$: the deeper network then contains every function of the shallower one. A plain layer $\mathbf{x} \mapsto \mathrm{ReLU}(\mathrm{BN}(\mathbf{W}\mathbf{x} + \mathbf{b}))$ can compute the identity on the nonnegative inputs that it receives, but only for particular values of its weights. So the function classes of plain networks are nested as well, and plain and residual networks both belong in the right panel of the figure. Nesting, then, is not what the plain networks lack. What the degradation results show is that gradient descent does not find those particular values, and nothing in the plain layer draws it toward them. A residual block changes the parameterization so that the identity lies at the simplest point of the parameter space, where the correction is zero.

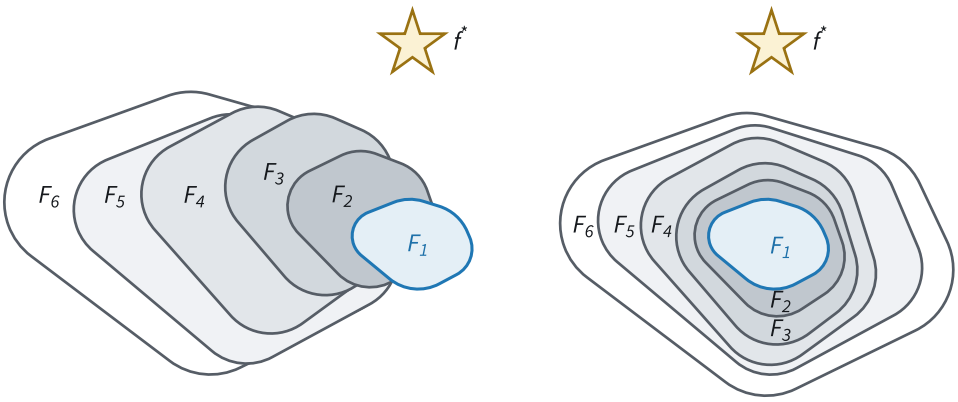


Figure 7.7: Function classes of increasing size. On the left the classes are not nested: a larger class need not contain the functions of a smaller one, and need not come closer to the target f^*. On the right each class contains the previous ones, so the best function available can only improve as the class grows.

7.4.2 Residual Blocks

Figure 7.8 compares a regular block with a residual one. Both receive the input $\mathbf{x}$ and feed their result into an activation function. The regular block on the left must learn the desired mapping $f(\mathbf{x})$ directly. The residual block on the right learns the *residual mapping* $g(\mathbf{x}) = f(\mathbf{x}) - \mathbf{x}$ and adds the input back, so its output is $\mathbf{x} + g(\mathbf{x})$. If the identity is the desired mapping, the residual block only has to drive g to zero, which it can do in its last layer alone, whatever the layers before it compute. The line that carries $\mathbf{x}$ to the addition is called a *residual connection* or *shortcut connection*.

The shortcut also changes how the gradient travels. Let us stack two such blocks, $\mathbf{x}_1 = \mathbf{x}_0 + g_1(\mathbf{x}_0)$ and $\mathbf{x}_2 = \mathbf{x}_1 + g_2(\mathbf{x}_1)$. Then $\mathbf{x}_2 = \mathbf{x}_0 + g_1(\mathbf{x}_0) + g_2(\mathbf{x}_1)$: the output contains the input itself, plus two corrections. Its Jacobian with respect to $\mathbf{x}_0$ is $(\mathbf{I} + \partial g_2/\partial \mathbf{x}_1)(\mathbf{I} + \partial g_1/\partial \mathbf{x}_0)$, which multiplies out to the identity plus terms that pass through one branch or through both. The same holds for any number of blocks, so the gradient of the loss reaches every block through the chain of identity terms, without passing through the convolutions of the blocks above it. A plain network has no such term: the gradient reaches a layer only through all the layers above it.

ResNet itself is not quite this clean. Two things remain on the identity path: the ReLU after each addition, which blocks the gradient wherever the sum is negative, and the 1×1 convolutions, with their batch normalization, on the shortcuts of the blocks that change the shape. The pre-activation design of the exercises removes the ReLU.

How much do these two arguments explain? They make it plausible that residual networks are easy to optimize, and Section 7.4.3 checks that they are. They do not show why plain networks fail. Vanishing gradients, the obvious suspect, are an unlikely cause: He et al. (2016) report that the gradients of their plain networks, which use batch normalization, kept a healthy size, and the networks degraded all the same. As with batch normalization, the remedy is better understood than the disease.

ResNet's residual block follows VGG's design of 3×3 convolutions. It has two 3×3 convolutional layers with the same number of output channels. Each convolution is followed by

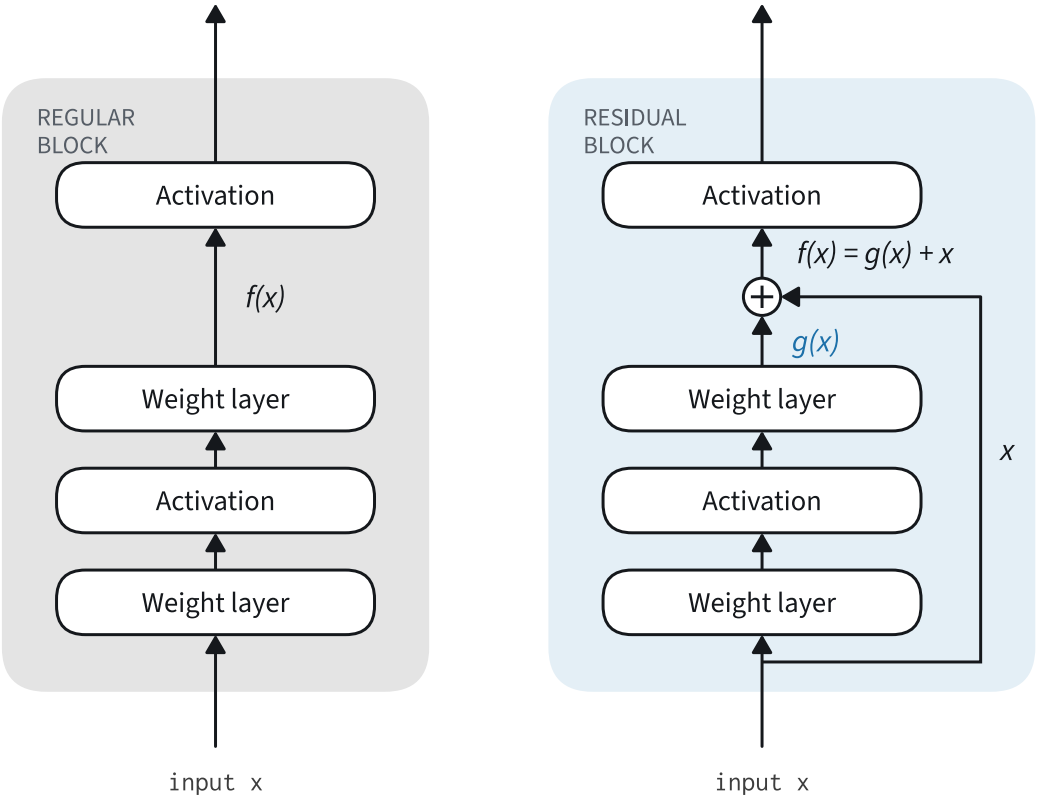


Figure 7.8: A regular block (left) must learn the mapping $f(\mathbf{x})$ directly. A residual block (right) learns the residual mapping $g(\mathbf{x}) = f(\mathbf{x}) - \mathbf{x}$ and adds the input back, so the identity mapping $f(\mathbf{x}) = \mathbf{x}$ corresponds to $g(\mathbf{x}) = 0$.

batch normalization, and the first also by a ReLU. The convolutions have no bias, since the batch normalization after them would remove it (Section 7.3). The block adds its input to the output of the second batch normalization and applies a final ReLU.

When is this block the identity? The last layer of its branch is a batch normalization, so the branch outputs zero when the scale γ and the shift β of that layer are zero. Because the final ReLU sets negative values to zero, the block is then the identity only on inputs that are already nonnegative. Every block of the network receives such inputs, since each one follows a ReLU. Note that the block does not start out this way: γ is initialized to 1, so the argument concerns what is easy to reach, not where training begins. Some later training recipes do start this γ at zero (He et al. 2019); we keep it at 1.

The output of the branch must have the shape of the input so that the two can be added. When a block changes the number of channels or halves the resolution, a 1×1 convolution with the same stride, followed by batch normalization, transforms the input on the shortcut to the new shape. This is rather crude: with stride 2, the convolution reads only one of every four input positions. An exercise repairs it. The class `Residual` implements both the identity and the 1×1 shortcut.

```
class Residual(nn.Module):
    """The Residual block of ResNet models."""
    def __init__(self, num_channels, use_1x1conv=False, strides=1):
        super().__init__()
        # No conv biases: the batch normalization after each conv removes them
        self.conv1 = nn.LazyConv2d(num_channels, kernel_size=3, padding=1,
                                   stride=strides, bias=False)
        self.conv2 = nn.LazyConv2d(num_channels, kernel_size=3, padding=1,
                                   bias=False)
        # Auto-enable 1x1 conv when downsampling so the residual shape matches.
        if use_1x1conv or strides != 1:
            self.conv3 = nn.LazyConv2d(num_channels, kernel_size=1,
                                       stride=strides, bias=False)
            self.bn3 = nn.LazyBatchNorm2d()
        else:
            self.conv3 = None
        self.bn1 = nn.LazyBatchNorm2d()
        self.bn2 = nn.LazyBatchNorm2d()

    def forward(self, X):
        Y = F.relu(self.bn1(self.conv1(X)))
        Y = self.bn2(self.conv2(Y))
        if self.conv3:
            X = self.bn3(self.conv3(X))
        Y += X
        return F.relu(Y)
```

Figure 7.9 shows the two variants: `use_1x1conv=True`, or a stride other than 1, selects the one on the right.

A block with matching shapes maps a minibatch of four three-channel 6×6 inputs to an output of the same shape.

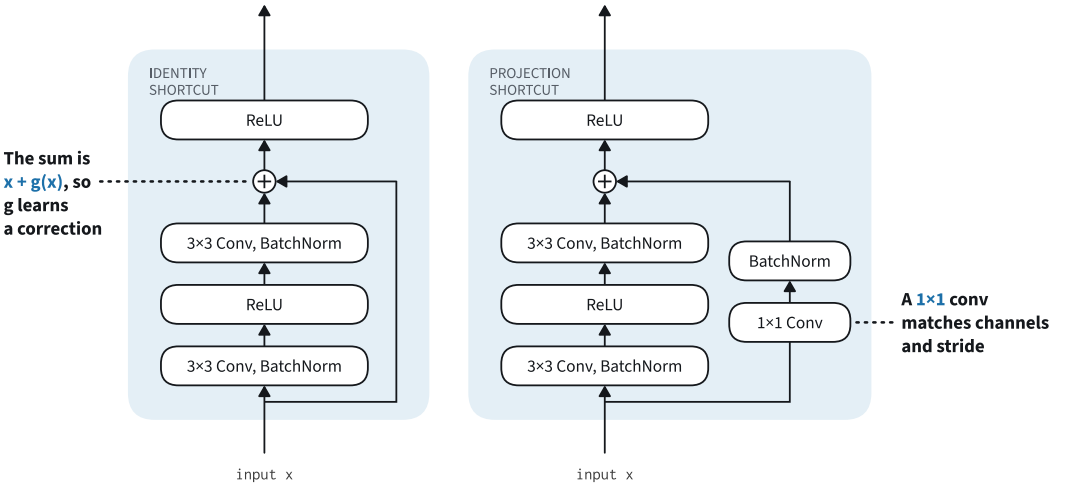


Figure 7.9: The two variants of the ResNet block: with an identity shortcut when input and output shapes match (left), and with a 1×1 convolution and batch normalization on the shortcut that adjust channels and resolution so that the addition is defined (right).

```
blk = Residual(3)
X = d2l.randn(4, 3, 6, 6)
blk(X).shape
```

```
torch.Size([4, 3, 6, 6])
```

With six output channels and stride 2, the block doubles the channels and halves the height and the width. This is the block that opens each later stage of ResNet below.

```
blk = Residual(6, use_1x1conv=True, strides=2)
blk(X).shape
```

```
torch.Size([4, 6, 3, 3])
```

7.4.3 Plain and Residual Networks Compared

Does the degradation problem appear on Fashion-MNIST, and does the residual block remove it? To find out we need networks that differ in the shortcut alone. We take the design that He et al. (2016) used on CIFAR-10 and apply it to the 28×28 images of Fashion-MNIST, at which depth is cheap. A *plain block* has the layers of `Residual` without the shortcut: two 3×3 convolutions, each followed by batch normalization and a ReLU.

```
def plain_block(num_channels, strides=1):
    """The layers of `Residual` without the shortcut."""
    return nn.Sequential(
        nn.LazyConv2d(num_channels, kernel_size=3, padding=1, stride=strides,
                      bias=False), nn.LazyBatchNorm2d(), nn.ReLU(),
        nn.LazyConv2d(num_channels, kernel_size=3, padding=1, bias=False),
```

```
        nn.LazyBatchNorm2d(), nn.ReLU())
```

The network starts with a 3×3 convolution with 16 output channels, batch normalization, and a ReLU. Three stages of n blocks follow, with 16, 32, and 64 channels; the first block of the second and of the third stage halves the height and the width. Global average pooling and a fully connected layer form the head. With two convolutional layers per block, the network has $6n + 2$ layers: 20 for $n = 3$ and 56 for $n = 9$. The argument `block` is `plain_block` or `Residual`. As in the earlier networks of this chapter, the weights start from He initialization (Section 4.4.2.2), and the learning rate follows the warmup and cosine schedule (`d2l.cosine_warmup` of Section 7.1). The peak is 0.1, the rate at which several networks of Section 7.2 were at the edge of stability. With a normalization after every convolution it is safe, as Section 7.3 led us to expect.

```
class DeepNet(d2l.Classifier):
    """6n + 2 layers: a 3x3 stem, three stages of n blocks, a linear head."""
    def __init__(self, block, n, lr=0.1, num_steps=2345, num_classes=10):
        super().__init__()
        self.save_hyperparameters(ignore=['block'])
        layers = [nn.LazyConv2d(16, kernel_size=3, padding=1, bias=False),
                  nn.LazyBatchNorm2d(), nn.ReLU()]
        for i, num_channels in enumerate((16, 32, 64)):
            for j in range(n):  # The first block of stages 2 and 3 downsamples
                strides = 2 if i > 0 and j == 0 else 1
                layers.append(block(num_channels, strides=strides))
        layers += [nn.AdaptiveAvgPool2d((1, 1)), nn.Flatten(),
                   nn.LazyLinear(num_classes)]
        self.net = nn.Sequential(*layers)

    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)

    def configure_optimizers(self):
        optim = torch.optim.SGD(self.parameters(), lr=self.lr)
        schedule = torch.optim.lr_scheduler.LambdaLR(optim, lambda t: (
            d2l.cosine_warmup(t, self.num_steps, self.num_steps // 10)))
        return optim, schedule  # The Trainer steps it after every minibatch
```

As for LeNet in Section 6.6, the Trainer calls the model's `weight_init` once the first minibatch has fixed the shapes of the lazy layers. It applies He initialization (`kaiming_normal_`) to every convolutional and fully connected layer, including those inside the `Residual` blocks.

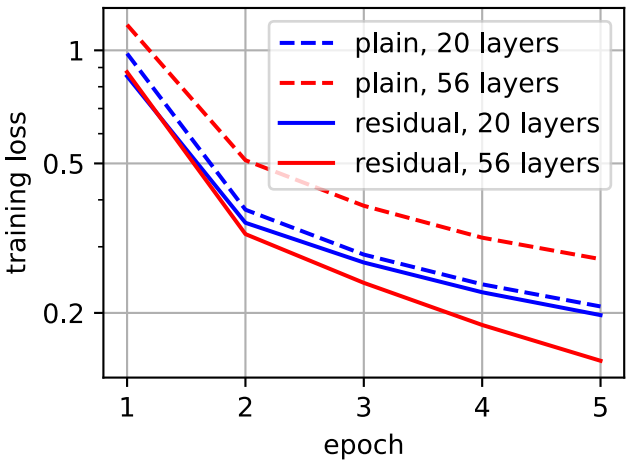
We train the plain and the residual network at both depths with identical settings, minibatches of 128 images and five epochs, and keep the history of each run.

```
data = d2l.FashionMNIST(batch_size=128)
num_steps = 5 * len(data.train_dataloader())
history = {}
```

```
for name, block in (('plain', plain_block), ('residual', Residual)):
    for n in (3, 9):
        model = DeepNet(block, n, lr=0.1, num_steps=num_steps)
        trainer = d2l.Trainer(max_epochs=5, num_gpus=1)
        trainer.board.display = False  # We plot the losses ourselves
        trainer.fit(model, data)
        history[f'{name}, {6 * n + 2} layers'] = trainer.history
```

The training losses of the four networks share one plot, on a logarithmic scale; the 20-layer networks are blue, the 56-layer ones red, and the plain networks dashed.

```
d2l.plot(list(range(1, 6)), [h['train_loss'] for h in history.values()],
         'epoch', 'training loss', legend=list(history), yscale='log',
         fmts=('b--', 'r--', 'b-', 'r-'))
```




```
{name: (round(h['train_loss'][-1], 3), round(h['val_acc'][-1], 3))
 for name, h in history.items()}  # Training loss, validation accuracy
```

```
{'plain, 20 layers': (0.208, 0.909),
 'plain, 56 layers': (0.278, 0.885),
 'residual, 20 layers': (0.197, 0.904),
 'residual, 56 layers': (0.149, 0.915)}
```

The plain networks show the degradation problem. The 56-layer one ends with a training loss of about 0.25 to 0.3, above the 0.2 or so of the 20-layer one, and its curve lies above the shallower network's in every epoch. It fits even its training data worse, so overfitting cannot be the cause. With residual blocks the order reverses: the 56-layer network ends at about 0.15 to 0.17, below the 20-layer one, which ends near 0.19 with or without shortcuts. At 20 layers the shortcut makes no visible difference. At 56 it decides whether 36 extra layers are a burden or a gain, and since the four networks differ in nothing but depth and the shortcut, there is no other explanation to reach for. Validation accuracy shows the same degradation more faintly: the deep plain network ends lowest, at 88% to 90%, while the other three end between 90.5% and 91.5%, too close together to order after five epochs.

7.4.4 ResNet-18

7.4.4.1 Stem, Stages, and Head

ResNet arranges residual blocks into the stem, body, and head of GoogLeNet (Section 7.2.3). The stem is GoogLeNet's: a 7×7 convolution with 64 output channels and stride 2, followed by 3×3 max-pooling with stride 2, which together reduce the resolution fourfold. ResNet adds batch normalization after the convolution. The method `stem` builds it.

```
class ResNet(d2l.Classifier):
    def stem(self):
        return 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))

    weight_init = DeepNet.weight_init  # He initialization
```

The body consists of four *stages*, each a sequence of residual blocks with the same number of output channels. The first stage keeps the 64 channels and the resolution of the stem, which has already reduced the resolution. The first block of each later stage doubles the number of channels and halves the height and the width. The method `stage` builds one stage, of `num_residuals` blocks.

```
@d2l.add_to_class(ResNet)
def stage(self, num_residuals, num_channels, first_block=False):
    blk = []
    for i in range(num_residuals):
        if i == 0 and not first_block:
            blk.append(Residual(num_channels, use_1x1conv=True, strides=2))
        else:
            blk.append(Residual(num_channels))
    return nn.Sequential(*blk)
```

The head is that of NiN and GoogLeNet: global average pooling, followed by a fully connected output layer. The argument `arch` lists the number of blocks and the number of channels of each stage. As in `DeepNet`, the weights start from He initialization. The only bias, that of the output layer, starts at zero.

```
@d2l.add_to_class(ResNet)
def __init__(self, arch, lr=0.1, num_steps=4690, num_classes=10):
    super(ResNet, self).__init__()
    self.save_hyperparameters()
    self.net = nn.Sequential(self.stem())
    for i, b in enumerate(arch):
        self.net.add_module(f'stage{i+1}', self.stage(*b, first_block=(i==0)))
    self.net.add_module('head', nn.Sequential(
        nn.AdaptiveAvgPool2d((1, 1)), nn.Flatten(),
        nn.LazyLinear(num_classes)))
```

With two blocks in each of the four stages we obtain ResNet-18. The name counts layers: each stage holds four 3×3 convolutional layers, not counting the 1×1 convolutions on the

shortcuts. Together with the 7×7 convolution of the stem and the fully connected output layer, these are 18 layers. ResNet-34 has 3, 4, 6, and 3 blocks per stage. ResNet-50, -101, and -152 replace the block by a *bottleneck* block of a 1×1, a 3×3, and a 1×1 convolution, the ResNeXt block below with a single group. Figure 7.10 shows ResNet-18.

```
class ResNet18(ResNet):
    def __init__(self, lr=0.1, num_steps=4690, num_classes=10):
        super().__init__(((2, 64), (2, 128), (2, 256), (2, 512)),
                         lr, num_steps, num_classes)
```

We follow the shape of a single-channel 96×96 image through the network. As before, the resolution shrinks and the number of channels grows until global average pooling collapses each channel to one number.

```
ResNet18().layer_summary((1, 1, 96, 96))
```

```
Sequential              (1, 64, 24, 24)
Sequential              (1, 64, 24, 24)
Sequential              (1, 128, 12, 12)
Sequential              (1, 256, 6, 6)
Sequential              (1, 512, 3, 3)
Sequential              (1, 10)
```

7.4.4.2 Training

We train ResNet-18 on Fashion-MNIST, upsampled to 96×96 pixels, with minibatches of 128 images and 10 epochs. It borrows the optimizer and the schedule of `DeepNet`, peak 0.1 included; with batch normalization in every block, that peak is safe.

```
ResNet.configure_optimizers = DeepNet.configure_optimizers  # Same schedule
```

```
data = d2l.FashionMNIST(batch_size=128, resize=(96, 96))
model = ResNet18(lr=0.1, num_steps=10 * len(data.train_dataloader()))
trainer = d2l.Trainer(max_epochs=10, num_gpus=1)
trainer.fit(model, data)
```

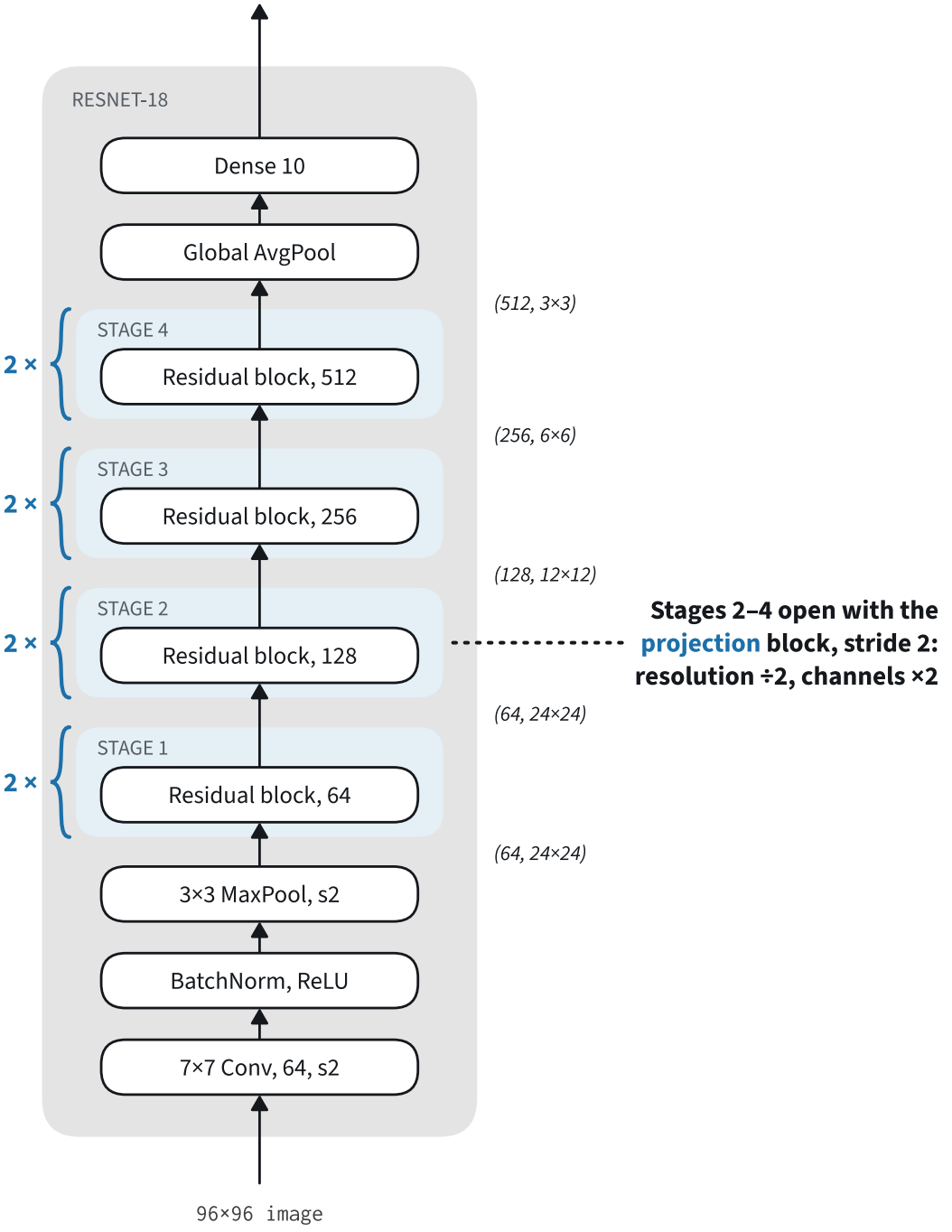
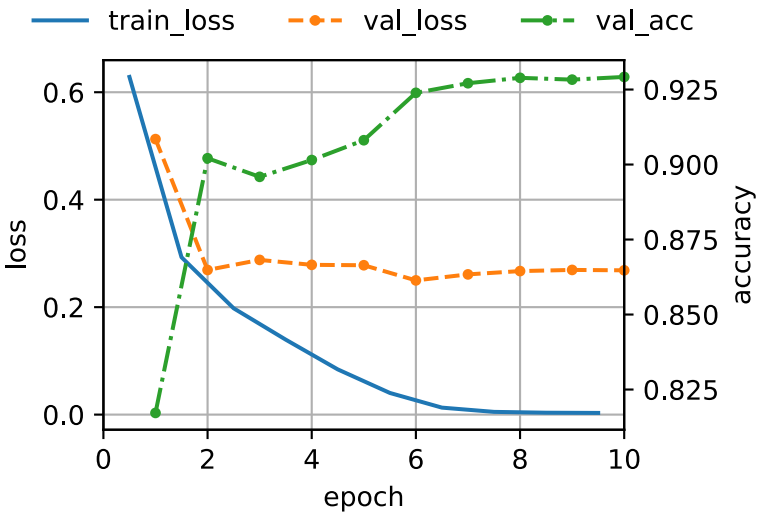


Figure 7.10: ResNet-18: a 7×7 convolutional stem, four stages of two residual blocks each, where the first block of stages 2, 3, and 4 halves the resolution and doubles the channels, and a head of global average pooling and one fully connected layer. Shapes are those of a 96×96 input.




```
round(trainer.history['val_acc'][-1], 3)  # Validation accuracy after 10 epochs
```

```
0.929
```

After 10 epochs ResNet-18 reaches a validation accuracy of about 93%, and its last three epochs change it by less than a point. That equals the quarter-width VGG, the best network of Section 7.2, with a quarter of its parameters and at 96×96 instead of 224×224 pixels. We should not read more into the comparison than it can bear. The two networks differ in architecture, normalization, and input size at once, so the run does not isolate what the residual connection contributes. The paired runs of Section 7.4.3 do.

The training loss of ResNet-18 falls below 0.01, far below its validation loss of about 0.27, which stops improving after the first few epochs: with 11.2 million parameters for 60,000 training images, the network fits the training data almost perfectly and new data much less well. More data, data augmentation, and regularization reduce this gap; the training recipes of Section 7.6 combine them.

7.4.5 ResNeXt and DenseNet

The residual block fixes how a block passes its input on: the input is added to the block's output. Two variants keep this idea and change the block. ResNeXt changes what the block computes, so that it can be wider at the same cost; DenseNet changes how the input is passed on, by concatenation instead of addition. We use the ResNeXt block again when we search over network designs in Section 7.8.

7.4.5.1 ResNeXt

More channels make a block more expressive, but the cost of a convolution that maps c_i input channels to c_o output channels grows with the product $c_\text{i}c_\text{o}$ (Section 6.4): doubling the width of a block quadruples its arithmetic. ResNeXt (Xie et al. 2017) borrows from the Inception block of Section 7.2.3, which processes its input in separate branches. Unlike Inception, it makes all branches identical.

A *grouped convolution* splits the c_i input channels into g groups of c_i/g channels and convolves each group separately into c_o/g output channels. Each group has a weight tensor of size $(c_\text{i}/g) \times (c_\text{o}/g) \times k \times k$, so the g groups together have $c_\text{i} c_\text{o} k^2/g$ weights instead of $c_\text{i} c_\text{o} k^2$, and the layer performs g times fewer multiply-adds. We assume that g divides both c_i and c_o. The measured speedup is usually smaller than g, since multiply-adds count arithmetic and not time (Section 7.5). Grouped convolutions first appeared in AlexNet (Krizhevsky et al. 2012), which split three of its five convolutional layers into two groups, one per GPU.

The groups of a grouped convolution exchange no information. The ResNeXt block in Figure 7.11 mixes them by placing the grouped 3×3 convolution between two 1×1 convolutions. The first reduces the c channels of the input to b *bottleneck* channels, and the second restores c channels. The two 1×1 convolutions have $2cb$ weights and the grouped 3×3 convolution $9b^2/g$. The shortcut is that of `Residual`. The stride of a downsampling block sits on the 3×3 convolution, not on the first 1×1 one.

What does "wider at the same cost" amount to? A bottleneck block in the first stage of ResNet-50 has $c = 256$, $b = 64$, and a single group, hence $2 \cdot 256 \cdot 64 + 9 \cdot 64^2 = 69{,}632$ weights. The block of Figure 7.11, with $b = 128$ and $g = 32$, has $2 \cdot 256 \cdot 128 + 9 \cdot 128^2/32 = 70{,}144$: twice the width in the middle for less than 1% more weights. On ImageNet this exchange lowers the top-1 error of ResNet-50 from 23.9% to 22.2% (Xie et al. 2017).

The class `ResNeXtBlock` takes the number of output channels `num_channels` (c), the number of groups `groups` (g), and `bot_mul`, the ratio b/c of bottleneck channels to output channels. As in `Residual`, `strides=2` halves the height and the width and switches on the 1×1 convolution on the shortcut.

```
class ResNeXtBlock(nn.Module):
    """The ResNeXt block."""
    def __init__(self, num_channels, groups, bot_mul, use_1x1conv=False,
                 strides=1):
        super().__init__()
        bot_channels = int(round(num_channels * bot_mul))
        # No conv biases: the batch normalization after each conv removes them
        self.conv1 = nn.LazyConv2d(bot_channels, kernel_size=1, stride=1,
                                   bias=False)
        self.conv2 = nn.LazyConv2d(bot_channels, kernel_size=3,
                                   stride=strides, padding=1,
                                   groups=groups, bias=False)
        self.conv3 = nn.LazyConv2d(num_channels, kernel_size=1, stride=1,
                                   bias=False)
        self.bn1 = nn.LazyBatchNorm2d()
        self.bn2 = nn.LazyBatchNorm2d()
        self.bn3 = nn.LazyBatchNorm2d()
        # Auto-enable 1x1 conv when downsampling so the residual shape matches.
        if use_1x1conv or strides != 1:
            self.conv4 = nn.LazyConv2d(num_channels, kernel_size=1,
                                       stride=strides, bias=False)
            self.bn4 = nn.LazyBatchNorm2d()
        else:
            self.conv4 = None

    def forward(self, X):
```

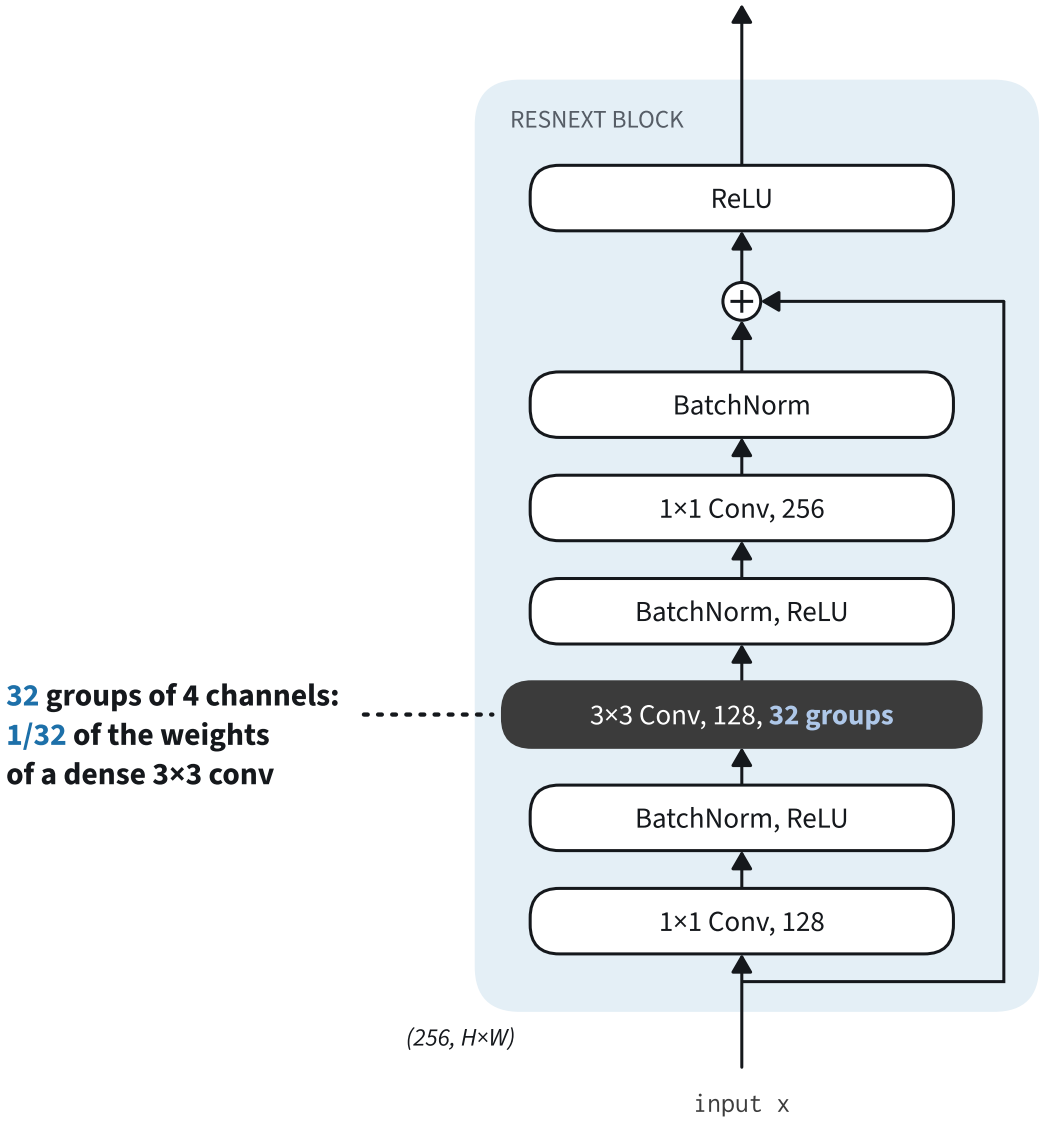


Figure 7.11: The ResNeXt block: a grouped 3×3 convolution with g groups between two 1×1 convolutions that mix information across the groups. The block shown has $c = 256$ channels, $b = 128$ bottleneck channels, and $g = 32$ groups of four channels.

```
        Y = F.relu(self.bn1(self.conv1(X)))
        Y = F.relu(self.bn2(self.conv2(Y)))
        Y = self.bn3(self.conv3(Y))
        if self.conv4:
            X = self.bn4(self.conv4(X))
        return F.relu(Y + X)
```

A block with 32 channels, 16 groups, and as many bottleneck channels as output channels (`bot_mul=1`) keeps the shape of its input.

```
blk = ResNeXtBlock(32, 16, 1)
X = d2l.randn(4, 32, 96, 96)
blk(X).shape
```

```
torch.Size([4, 32, 96, 96])
```

7.4.5.2 DenseNet

A residual block computes $\mathbf{x}+g(\mathbf{x})$: its output keeps the number of channels of its input, and what the block adds is mixed into the input. *DenseNet* (dense convolutional network) (Huang et al. 2017) instead *concatenates* the output of each layer to its input along the channel axis. Where a stack of residual blocks computes $\mathbf{x}_k = \mathbf{x}_{k-1} + g_k(\mathbf{x}_{k-1})$, a sequence of dense layers $f_1, f_2, \ldots$ computes

$$\mathbf{x}_k = \left[\mathbf{x}_{k-1}, f_k(\mathbf{x}_{k-1})\right],$$

where $[\cdot,\cdot]$ denotes concatenation. Nothing is ever overwritten: $\mathbf{x}_k$ holds the input and the outputs of all k layers so far, so every layer is connected to all its predecessors, hence the name. Figure 7.12 shows a dense block with two layers.

Each convolution in a dense block is preceded by batch normalization and a ReLU, the "normalization, activation, convolution" order that the exercises explore for ResNet. The convolutions have no bias, since every layer that reads their output starts with batch normalization. The number of channels that each layer appends, 32 in Figure 7.12, is called the *growth rate*. Since every dense block adds channels, a stack of them would soon be very wide. Between dense blocks, a *transition layer* therefore reduces the number of channels with a 1×1 convolution and halves the height and the width with average pooling of stride 2. A full DenseNet has four dense blocks separated by transition layers, like the four stages of ResNet-18; problem 9 builds and trains one.

DenseNet reached the ImageNet accuracy of ResNet with fewer parameters, which its authors attribute to the reuse of features. Unfortunately, concatenation costs memory. The distinct feature maps of a dense block grow linearly with its depth, but a direct implementation materializes a new, wider concatenation for every layer and keeps it, with its normalized version, for the backward pass, so its memory grows quadratically with the depth of the block. Memory-efficient implementations recompute these tensors during backpropagation (Pleiss et al. 2017). Addition has no such cost, which is one reason why residual connections, not dense ones, became the standard in later architectures.

Concatenating skip connections did survive in one widely used design. The U-Net (Ronneberger et al. 2015), a network for image segmentation, first reduces the resolution step

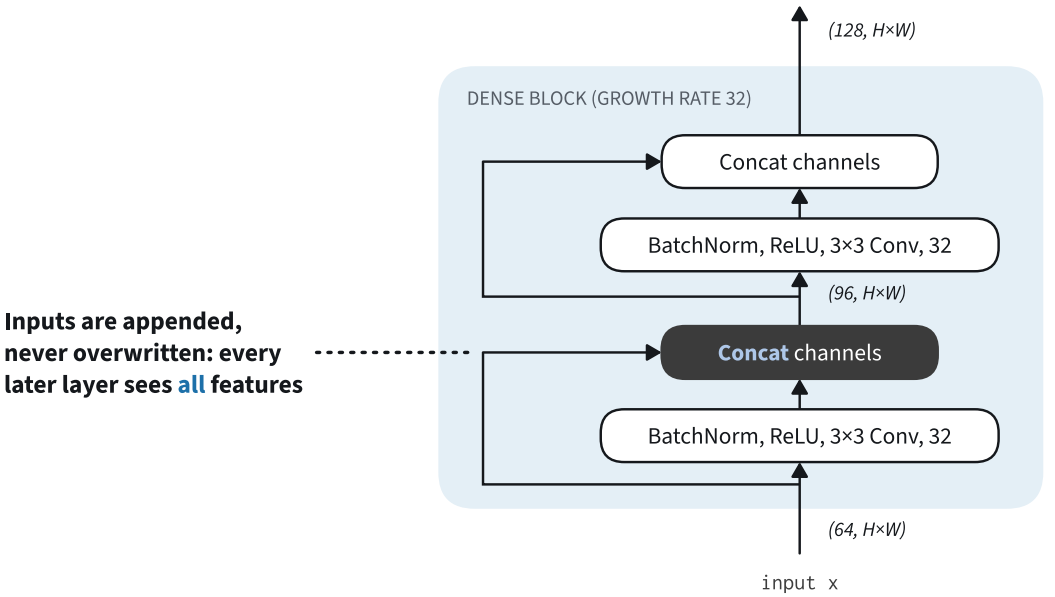


Figure 7.12: A dense block with two layers and growth rate 32. Each layer appends its 32 output channels to its input, so a 64-channel input grows to 96 and then 128 channels, and every layer sees the outputs of all layers before it.

by step and then restores it. On the way up it concatenates each upsampled feature map with the feature map of the same resolution from the way down. It does so once per resolution rather than after every layer, so its memory stays modest. The denoising networks of diffusion models are built on it (Section 17.3).

7.4.6 Summary

Deeper plain networks can fit even their training data worse than shallower ones, although they could represent the same functions; why training fails to find them is not fully understood. A residual block computes its input plus a learned correction, which moves the identity from particular weight values to the point of zero correction and gives the gradient a direct path to every block. On Fashion-MNIST, a plain network with 56 layers ends training with a higher training loss than one with 20 layers, while with residual blocks the deeper network ends with the lower one. Stacked in four stages with batch normalization, residual blocks give ResNet-18, which reaches about 93% validation accuracy on Fashion-MNIST in 10 epochs, and the same template scales to 152 layers. ResNeXt replaces the middle convolution of a bottleneck block by a grouped convolution, which makes the block wider at the same cost, and DenseNet concatenates the outputs of its layers instead of adding them, which reuses features but needs more memory. Residual connections went on to become a part of nearly every deep architecture, including the transformers of Chapter 11.

7.4.7 Exercises

1. **Inception versus residual blocks.** In the `arch` of GoogLeNet (Section 7.2.3), the second block of the middle group has branch widths $c_1 = 160$, $c_2 = (112, 224)$,

$c_3 = (24, 64)$, and $c_4 = 64$, and it maps 512 input channels to 512 output channels. Compare it with a 512-channel `Residual` block that uses an identity shortcut.

1. Count the convolution weights and the multiply-adds per output pixel of both blocks, ignoring biases and normalization parameters.
2. A `Residual` block whose residual branch outputs zero is the identity on nonnegative inputs. Determine whether any choice of parameters makes the Inception block the identity on nonnegative inputs, and state what the answer implies for deepening GoogLeNet by inserting Inception blocks.

2. **Pre-activation ordering.** He et al. (2016)*1 moved batch normalization and the ReLU in front of each convolution of the residual branch and removed the ReLU after the addition (their Figure 1), so that a block computes $\mathbf{x} + g(\mathbf{x})$ with the identity path left untouched. The DenseNet convolution block in Section 7.4.5.2 uses this "normalization, activation, convolution" order.

   1. Stack K pre-activation blocks, $\mathbf{x}_k = \mathbf{x}_{k-1} + g_k(\mathbf{x}_{k-1})$ for $k = 1, \dots, K$. Write $\mathbf{x}_K$ in terms of $\mathbf{x}_0$ and the branch outputs, and show that the Jacobian $\partial \mathbf{x}_K / \partial \mathbf{x}_0$ contains the identity matrix as an additive term.
   2. Now stack K `Residual` blocks with identity shortcuts, each of which applies a ReLU after the addition. Which factor takes the place of the identity term in $\partial \mathbf{x}_K / \partial \mathbf{x}_0$, and what does it imply for the gradient that reaches the first block through the shortcuts?

3. **Nested function classes.** The containment argument of Figure 7.7 applies to a deeper network when each added block can represent the identity on every input it receives. Every `Residual` block in `ResNet18` receives nonnegative inputs.

   1. Set the weights of `conv2` in a `Residual` block with an identity shortcut to zero. Determine the output of `bn2` in training mode and in prediction mode, and decide in each mode whether the block is then the identity on nonnegative inputs. Name parameter values of `bn2` that make the block the identity in both modes.
   2. ResNet-34 (Table 1 of He et al. (2016)) has the same stage widths as `ResNet18` and more blocks per stage. Decide whether ResNet-34 represents every function that `ResNet18` represents, and justify the answer block by block.

4. code **Gradients of plain networks.** ⊠ According to He et al. (2016), the gradients of their plain networks kept a healthy size. Check this on the four networks that Section 7.4.3 trains.

   1. At initialization, compute the gradient of the loss on one minibatch and plot the norm of the gradient of each convolutional layer's weights against the index of the layer, for all four networks. Do the gradients of the 56-layer plain network shrink toward the input?
   2. Repeat the measurement after one epoch of training.
   3. Train the 56-layer plain network for 20 epochs instead of 5. Does its training loss catch up with that of the 20-layer plain network trained equally long? What does the answer say about whether the deep plain network is slow or stuck?

5. **Paths through a residual stack.** A stack of K residual blocks can be unraveled into an ensemble of paths: on each block, the signal either passes through the block's body or skips it (Veit et al. 2016).

   1. How many end-to-end paths pass through exactly ℓ of the K bodies? Verify your

formula by explicit enumeration for $K = 4$.
2. Relate the distribution of path lengths to why very deep residual networks remain trainable where equally deep plain networks are not.

Adapted from Simon Prince, Understanding Deep Learning, Problem 11.2.

6. code **ResNet variants.** ▣ Table 1 of He et al. (2016) specifies ResNet-18 to ResNet-152 by the number of blocks per stage and by the block type.
   1. Build ResNet-34 by passing its block counts as `arch` to `ResNet` and report its parameter count against `ResNet18`.
   2. ResNet-50 and its deeper relatives use a bottleneck block: a 1×1 convolution that reduces the width fourfold, a 3×3 convolution, and a 1×1 convolution that restores the width. `ResNeXtBlock` with `groups=1` and `bot_mul=0.25` implements this block. Compare its parameter count and multiply-adds with those of `Residual` at the same input and output width, and with those of `Residual` at a quarter of that width, which is the width of the matching ResNet-34 stage.
   3. Override `stage` in a subclass of `ResNet` so that it stacks bottleneck blocks, build ResNet-50, and compare its parameter count with ResNet-34's.

7. code **ResNet-D shortcut.** In a downsampling `Residual` block, the stride-2 1×1 convolution on the shortcut reads one of every four input positions. He et al. (2019) replace it by 2×2 average pooling with stride 2, followed by a stride-1 1×1 convolution and batch normalization.
   1. Implement this shortcut in a subclass of `Residual` and check on a random input that the new block has the same output shape and the same number of parameters as the original one.
   2. Build a ResNet-18 from the new block and train it with the settings of this section. Compare its validation accuracy with that of `ResNet18` over several runs of each network. Is the difference larger than the variation between runs?

8. code **Pre-activation ResNet-18.** ▣ Implement the pre-activation block of He et al. (2016)*1 as a variant of `Residual`, override `stage` in a subclass of `ResNet` so that it stacks the new block, and compare the training curves of the resulting ResNet-18 with those of `ResNet18`.

9. code **DenseNet.** ▣ Build and train the DenseNet of Section 7.4.5.2, then account for its parameters and its memory.
   1. Implement the convolution block of a dense block (batch normalization, a ReLU, and a 3×3 convolution without bias), a `DenseBlock` that applies `num_convs` such blocks and concatenates the output of each one to its input along the channel axis, and a transition layer (batch normalization, a ReLU, a 1×1 convolution without bias, and 2×2 average pooling with stride 2). Check on a random input with 3 channels that a dense block with two convolutions and growth rate 10 returns $3 + 10 + 10 = 23$ channels, and that a transition layer with 10 channels reduces them to 10 and halves the height and the width.
   2. Assemble a DenseNet from the stem of `ResNet`, four dense blocks of four convolutions each with growth rate 32, a transition layer that halves the number of channels between consecutive dense blocks, and batch normalization and a ReLU before the head of `ResNet`. Train it with the settings of this section and compare its parameter count, validation accuracy, and time per epoch with those of `ResNet18`.

3. Huang et al. (2017) report that DenseNets match the accuracy of comparable ResNets with fewer parameters, whereas Pleiss et al. (2017) had to devise memory-efficient implementations for them. Consider a dense block whose n convolutions each emit r channels on an input with c channels, and compare it with n `Residual` blocks of constant width c. Ignore biases and batch normalization parameters. Your `DenseBlock` builds the input of each convolution as a new concatenated tensor, and a naive implementation keeps all of these inputs for the backward pass. How many channels does the i-th convolution consume? Sum them over the block and compare the sum with the $2n$ inputs of width c that the 3×3 convolutions of the residual blocks keep.
4. Count the weights of the dense block's 3×3 convolutions and compare them with those of the n residual blocks. For which growth rates r does the dense block use fewer parameters?
5. Pleiss et al. (2017) store only the $c + nr$ channels of distinct feature maps and discard each concatenated tensor and its normalized version after use. For $c = 64$, $r = 32$, and $n = 6$, compare this storage with your sum from the third part. Explain why the backward pass must then recompute concatenations, batch normalizations, and ReLUs, but no convolution.

7.5 Efficient Convnets

The networks of the previous sections were designed to win accuracy benchmarks, and they were trained and run on datacenter GPUs. Many convolutional networks run elsewhere: on phones, cameras, cars, and embedded boards. There latency, memory, and energy are fixed budgets, and a network that exceeds them cannot ship, however accurate it is. A phone that classifies every frame its camera records is a simple example. VGG-16 needs 15.3 billion multiply-adds to classify one 224×224 image, so at 30 frames a second the phone would have to perform 460 billion multiply-adds per second. The network's 138 million parameters would occupy 550 MB in single precision, in memory next to every other app. How can we make a convolutional network an order of magnitude cheaper, and what does that cost in accuracy?

This section studies three ideas that make convnets cheaper. *Factorization* replaces each dense convolution by a per-channel spatial filter and a 1×1 channel mixer. *Scaling and search* fixes the block and lets an optimizer choose the widths, depths, and input resolution. *Re-parameterization* trains a network with several branches per layer and then merges the branches into a single convolution that computes the same function. We also measure what the first idea delivers, in accuracy and in time. The answer is less flattering than the count of multiply-adds, and it depends on the task.

```
import time
```

7.5.1 Depthwise-Separable Networks

Most of the arithmetic of a convnet is spent in its dense convolutions, so the first idea is to make each of them cheaper.

7.5.1.1 Depthwise-Separable Convolutions

The cost of a dense convolution grows with the product $c_{\mathrm{i}}c_{\mathrm{o}}$ because every output channel reads every input channel. The grouped convolution of ResNeXt (Section 7.4.5.1) relaxes this: it splits the channels into g groups and lets each output group read only its own input group, which divides parameters and multiply-adds by g. Figure 7.13 shows which input channels each output channel reads: the dense pattern becomes block diagonal.

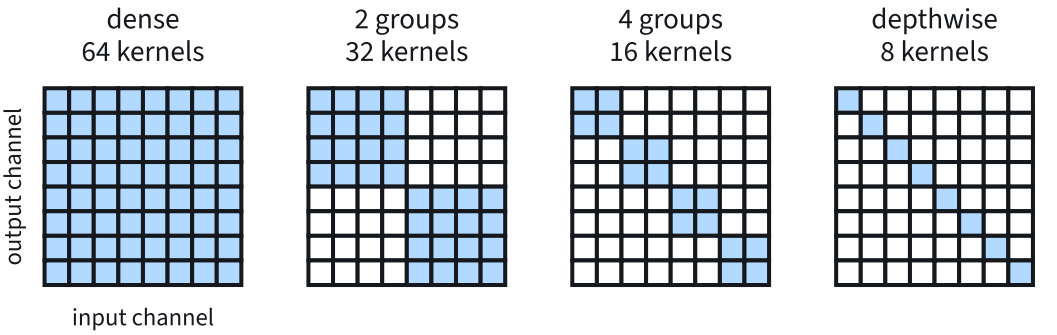


Figure 7.13: Input channels read by each output channel, for 8 input and 8 output channels. Each shaded cell is one $k_{\mathrm{h}} \times k_{\mathrm{w}}$ kernel; g groups keep $1/g$ of them.

Let us take grouping to its extreme. With $g = c_{\mathrm{i}} = c_{\mathrm{o}}$, every group holds one channel. This *depthwise convolution* filters each channel with its own $k \times k$ kernel and mixes no channels at all, the opposite of the 1×1 convolution (Section 6.4.3), which mixes channels and ignores neighbors. Each does half of what a dense convolution does, and together they do both: a depthwise $k \times k$ convolution followed by a *pointwise* 1×1 convolution is a *depthwise-separable convolution* (Chollet 2017; Howard et al. 2017). Figure 7.14 contrasts it with the dense layer it replaces: the depthwise stage looks at neighborhoods within each channel, and the pointwise stage recombines the channels at each position.

For a $k \times k$ kernel and an $h \times w$ output, the dense convolution performs $hw \cdot c_{\mathrm{i}}c_{\mathrm{o}}k^2$ multiply-adds. The separable pair performs $hw \cdot c_{\mathrm{i}}k^2$ in the depthwise stage and $hw \cdot c_{\mathrm{i}}c_{\mathrm{o}}$ in the pointwise stage, so the cost relative to the dense layer is

$$\frac{hw \cdot c_{\mathrm{i}}k^2 + hw \cdot c_{\mathrm{i}}c_{\mathrm{o}}}{hw \cdot c_{\mathrm{i}}c_{\mathrm{o}}k^2} = \frac{1}{c_{\mathrm{o}}} + \frac{1}{k^2}. \tag{7.6}$$

The same ratio holds for the weights. In Figure 7.14, with $c_{\mathrm{i}} = c_{\mathrm{o}} = k = 3$, the dense layer has $3 \cdot 27 = 81$ weights and the separable pair $27 + 9 = 36$, a ratio of $1/3 + 1/9 = 4/9$. Each further output channel costs the dense layer another 27 weights and the separable pair another 3. For $k = 3$ and many output channels the ratio is close to $1/9$: the separable layer is nearly nine times cheaper.

The savings come from a restriction. The separable pair applies the depthwise filter $\mathbf{D}_c$ to input channel c and scales the result by the pointwise weight $P_{o,c}$ before adding it to output channel o. It is therefore the dense convolution with kernel $\mathbf{K}_{o,c} = P_{o,c}\,\mathbf{D}_c$: all output channels see the same spatial filter of each input channel, only scaled, while a dense layer may use a different filter for every pair (o, c). Efficient architectures accept this restriction and use the savings for more channels or more layers, as the rest of this section shows.

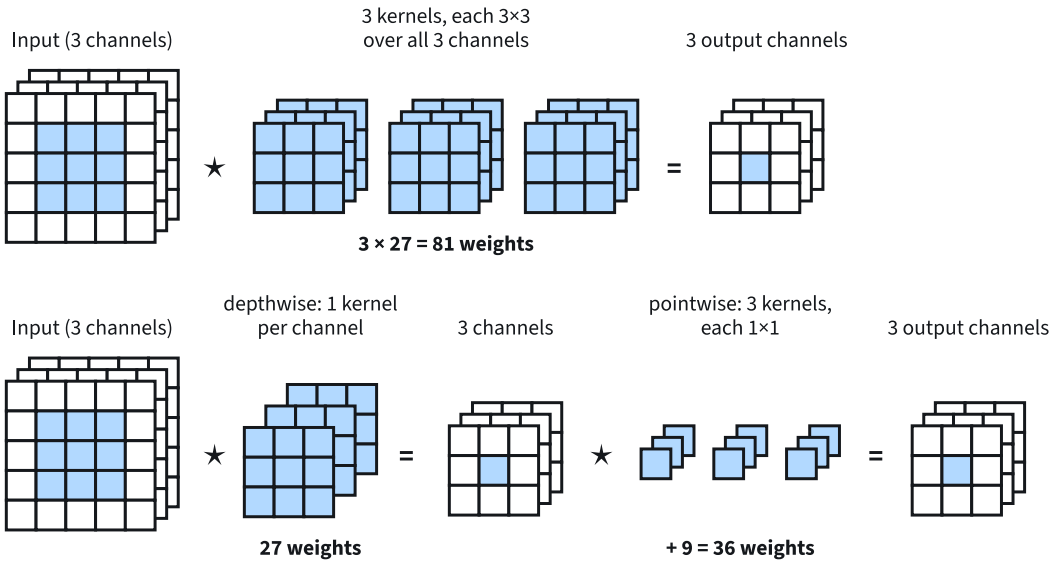


Figure 7.14: Three input and three output channels with 3×3 kernels. Top: a dense convolution, in which every output channel reads every input channel through a kernel stack of its own. Bottom: a depthwise convolution filters each channel separately, once for all output channels, and a pointwise convolution mixes the channels with one 1×1 kernel stack per output channel.

We check the arithmetic with a dense 3×3 convolution from 128 to 128 channels and its depthwise-separable factorization, without biases. Both map the input to an output of the same shape, and Equation 7.6 predicts a ratio of parameter counts of $(1/128 + 1/9)^{-1} \approx 8.4$.

```
c_i, c_o, k = 128, 128, 3
X = d2l.normal(0, 1, (1, c_i, 32, 32))
dense = nn.LazyConv2d(c_o, kernel_size=k, padding=1, bias=False)
depthwise = nn.LazyConv2d(c_i, kernel_size=k, padding=1, groups=c_i,
                          bias=False)
pointwise = nn.LazyConv2d(c_o, kernel_size=1, bias=False)
assert dense(X).shape == pointwise(depthwise(X)).shape
p_dense = dense.weight.numel()
p_sep = depthwise.weight.numel() + pointwise.weight.numel()
p_dense, p_sep, p_dense / p_sep
```

```
(147456, 17536, 8.408759124087592)
```

The dense layer has 147,456 weights and the separable pair 17,536, a ratio of 8.4 as predicted.

In PyTorch, `groups=c_i` turns `nn.Conv2d` into a depthwise convolution; any `groups` that divides both channel counts gives a grouped convolution.

7.5.1.2 MobileNet

MobileNet (Howard et al. 2017) turned this layer into a whole network. Every layer after the first is the block of Figure 7.15: a depthwise 3×3 convolution and a pointwise 1×1 convolution, each followed by batch normalization and a ReLU. The network downsamples by giving the depthwise convolution a stride of 2, so it needs no pooling layers inside, and it ends with global average pooling and one linear layer, as NiN does (Section 7.2.2). This answers our opening question rather well. On ImageNet, MobileNet comes within one point of VGG-16's accuracy (70.6% against 71.5%) with 0.57 billion multiply-adds and 4.2 million parameters: 27 times less arithmetic and 33 times fewer parameters.

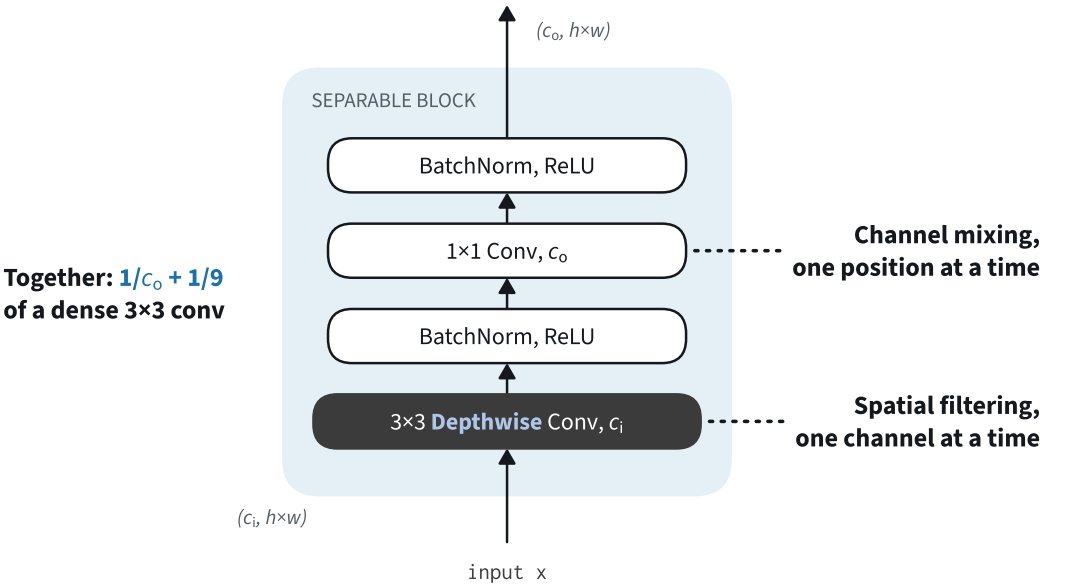


Figure 7.15: A depthwise-separable block from c_i to c_o channels. The depthwise 3×3 convolution filters each input channel separately; the pointwise 1×1 convolution mixes the channels at each position; batch normalization and a ReLU follow each convolution.

7.5.1.3 The Inverted Bottleneck

The bottleneck blocks of ResNet and ResNeXt (Section 7.4.5.1) compress: a 1×1 convolution reduces a wide representation, the 3×3 convolution works in the narrow space, and a final 1×1 convolution expands it back. This is sensible as long as the 3×3 convolution is dense and therefore expensive. A depthwise convolution is cheap even on wide tensors, and it has a reason to want them: it learns one spatial filter per channel, so the number of filters in a block equals the width at which the depthwise layer runs.

MobileNetV2 (Sandler et al. 2018) therefore *inverts* the bottleneck. The residual stream stays thin, say 24 channels. A 1×1 convolution expands it sixfold to 144 channels, the depthwise 3×3 convolution filters in the wide space, and a 1×1 projection returns to 24 channels. No ReLU follows the projection: applied to so thin a representation, a ReLU would zero the negative coordinates and with them information that later layers cannot recover. Figure 7.16 places the two blocks side by side.

The residual connection joins the thin ends, so the tensors that stay in memory from one

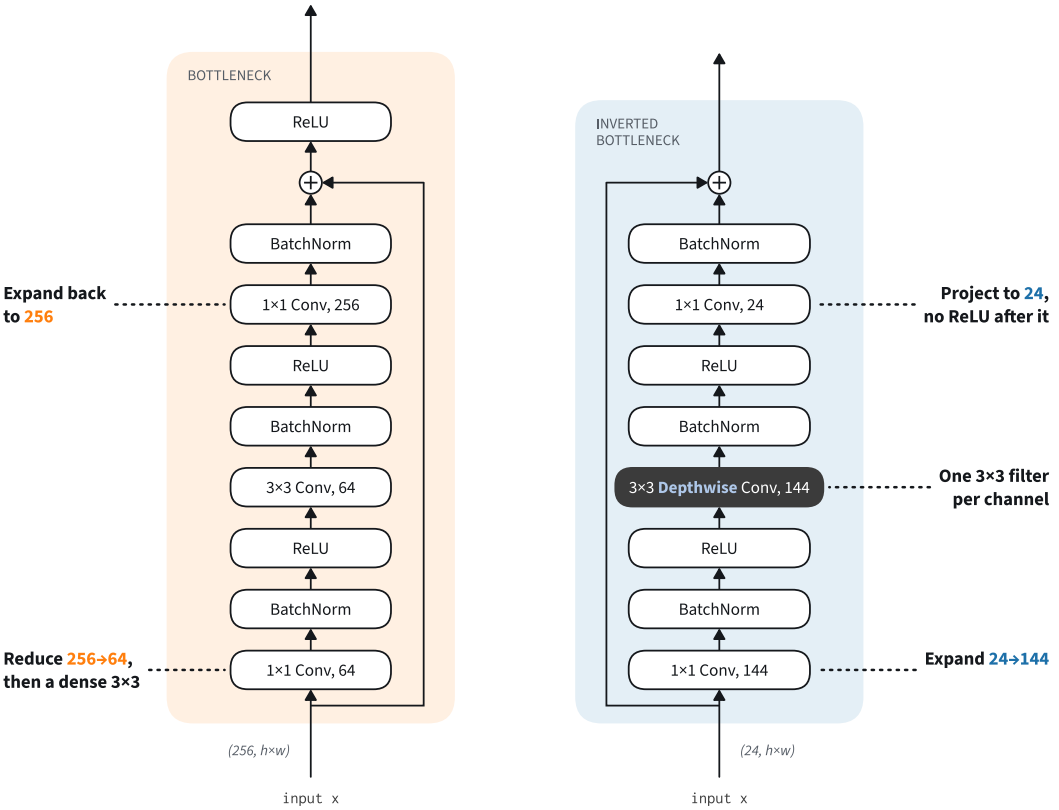


Figure 7.16: A ResNet bottleneck block (left) and a MobileNetV2 inverted bottleneck (right); the input is at the bottom. The bottleneck narrows 256 channels to 64 for a dense 3×3 convolution and widens them again. The inverted bottleneck widens 24 channels to 144 for a depthwise 3×3 convolution, which learns one filter per channel, and projects back to 24 without a ReLU; the residual connection joins the thin ends.

block to the next are the small ones. This matters on a phone, where activation memory rather than parameter storage is often the binding constraint. The same block shape, with a 7×7 depthwise convolution moved to the front, reappears in ConvNeXt (Section 7.7).

7.5.1.4 A Mini-MobileNet

We build a small MobileNetV1-style network for Fashion-MNIST. We use the original block rather than the inverted bottleneck because it shows the factorization most directly. The function below builds the pair of Figure 7.15; a stride of 2 downsamples. Every convolution in the networks of this section is followed by batch normalization, which has its own shift, so the convolutions carry no bias. All convolution and linear weights start from He initialization (Section 4.4).

```
def dws_block(in_channels, out_channels, stride=1):
    """Depthwise 3x3 and pointwise 1x1 convolutions, each with BN and ReLU."""
    return nn.Sequential(
        nn.Conv2d(in_channels, in_channels, kernel_size=3, stride=stride,
                  padding=1, groups=in_channels, bias=False),
        nn.BatchNorm2d(in_channels), nn.ReLU(),
        nn.Conv2d(in_channels, out_channels, kernel_size=1, bias=False),
        nn.BatchNorm2d(out_channels), nn.ReLU())

def he_init(module):
    """Normal weights with variance 2/n_in and zero biases."""
    if isinstance(module, (nn.Conv2d, nn.Linear)):
        nn.init.kaiming_normal_(module.weight)
        if module.bias is not None:
            nn.init.zeros_(module.bias)
```

Like VGG (Section 7.2.1), the whole network is a list of block parameters, here (channels, stride) pairs. A 3×3 convolution with stride 2 lifts the grayscale input to 32 channels at half resolution; seven depthwise-separable blocks then double the channels three times while halving the feature map three times; global average pooling and a linear layer produce the class scores.

The optimizer is new to this chapter. We train with stochastic gradient descent with momentum 0.9, which steps along an exponentially weighted sum of recent gradients instead of the latest gradient alone, so that its steps are up to ten times longer than plain gradient steps at the same rate (Section 9.4). The accuracies below are therefore not comparable with those of ResNet-18 in Section 7.4, which was trained with plain SGD. The learning rate again decays to zero along a half cosine (Section 7.1), here without warmup and with the scheduler that each framework provides: it starts at `lr` and reaches zero after the `num_steps` minibatches of training (4,690 for 10 epochs).

```
class MiniMobileNet(d2l.Classifier):
    def __init__(self, arch=((64, 1), (128, 2), (128, 1), (256, 2), (256, 1),
                             (512, 2), (512, 1)),
                 lr=0.05, num_steps=4690, num_classes=10):
        super().__init__()
        self.save_hyperparameters()
        layers = [nn.Conv2d(1, 32, kernel_size=3, stride=2, padding=1,
```

```
                          bias=False),
                nn.BatchNorm2d(32), nn.ReLU()]
        c = 32
        for c_out, stride in arch:
            layers.append(dws_block(c, c_out, stride))
            c = c_out
        layers += [nn.AdaptiveAvgPool2d((1, 1)), nn.Flatten(),
                   nn.Linear(c, num_classes)]
        self.net = nn.Sequential(*layers)
        self.net.apply(he_init)

    def configure_optimizers(self):
        optim = torch.optim.SGD(self.parameters(), lr=self.lr, momentum=0.9)
        return optim, torch.optim.lr_scheduler.CosineAnnealingLR(
            optim, self.num_steps)  # Stepped after every minibatch
```

How many parameters does the network have, and how many would it need with dense convolutions? Counting the trainable parameters answers the first question; we save the function `num_params` for later sections.

```
def num_params(model):
    """The number of trainable parameters."""
    return sum(p.numel() for p in model.parameters() if p.requires_grad)

model = MiniMobileNet()
assert model(torch.zeros(1, 1, 96, 96)).shape == (1, 10)
num_params(model)
```

```
542474
```

The network has 542,474 parameters. A twin that replaced each separable pair by one dense 3×3 convolution between the same channel counts would need $9 \sum_l c_{\mathrm{i},l}\, c_{\mathrm{o},l} = 4{,}663{,}296$ weights in its seven blocks alone, 8.8 times the 530,528 weights of the separable convolutions. With up to 512 output channels, the network comes close to the limit of 9.

7.5.1.5 Arithmetic, Accuracy, and Time

Does the factorization give up accuracy? To find out, we compare the mini-MobileNet with a dense VGG-style network (Section 7.2.1) that has about the *same number of parameters*. A dense 3×3 layer between the same two widths has about nine times as many weights, so with the same budget the dense network stops at 128 channels where the separable one reaches 512. Both networks use batch normalization after every convolution, the same initialization, the same global-average-pooling head, the same optimizer, and the same training budget. The comparison therefore matches parameters, not architectures: the separable block and the wider channels it allows change together.

```
class VGGSmall(d2l.Classifier):
    """A dense VGG-style network with the same parameter budget."""
    def __init__(self, arch=((2, 32), (2, 64), (2, 128), (2, 128)),
                 lr=0.05, num_steps=4690, num_classes=10):
```

```
        super().__init__()
        self.save_hyperparameters()
        layers, c = [], 1
        for num_convs, c_out in arch:
            for _ in range(num_convs):
                layers += [nn.Conv2d(c, c_out, kernel_size=3, padding=1,
                                     bias=False),
                           nn.BatchNorm2d(c_out), nn.ReLU()]
                c = c_out
            layers.append(nn.MaxPool2d(2))
        layers += [nn.AdaptiveAvgPool2d((1, 1)), nn.Flatten(),
                   nn.Linear(c, num_classes)]
        self.net = nn.Sequential(*layers)
        self.net.apply(he_init)

    def configure_optimizers(self):
        optim = torch.optim.SGD(self.parameters(), lr=self.lr, momentum=0.9)
        return optim, torch.optim.lr_scheduler.CosineAnnealingLR(
            optim, self.num_steps)  # Stepped after every minibatch
```

Before training, we count the arithmetic. A convolution with "same" padding and stride s turns an $h \times h$ input into an output of side $\lceil h/s \rceil$, and each output position of each output channel costs $k^2 c_\text{i}/g$ multiply-adds for a convolution with g groups. The functions below add these costs up for both networks on 96×96 inputs, including the linear head.

```
def conv_macs(h, c_in, c_out, k=3, stride=1, groups=1):
    """Output side and multiply-adds of a k x k convolution, 'same' padding."""
    h = -(-h // stride)  # ceil(h / stride)
    return h, h * h * k * k * c_in * c_out // groups

def mobilenet_macs(arch, h=96, num_classes=10):
    h, total = conv_macs(h, 1, 32, stride=2)  # The stem
    c = 32
    for c_out, stride in arch:
        h, depthwise = conv_macs(h, c, c, stride=stride, groups=c)
        h, pointwise = conv_macs(h, c, c_out, k=1)
        total, c = total + depthwise + pointwise, c_out
    return total + c * num_classes

def vgg_macs(arch, h=96, num_classes=10):
    total, c = 0, 1
    for num_convs, c_out in arch:
        for _ in range(num_convs):
            h, macs = conv_macs(h, c, c_out)
            total, c = total + macs, c_out
        h //= 2  # 2 x 2 max-pooling
    return total + c * num_classes

narrow = ((2, 11), (2, 23), (2, 46), (2, 46))  # VGGSmall's widths / 2.8
macs = {'Mini-MobileNet': mobilenet_macs(MiniMobileNet().arch),
        'VGG-style': vgg_macs(VGGSmall().arch),
        'VGG-style, narrow': vgg_macs(narrow)}
```

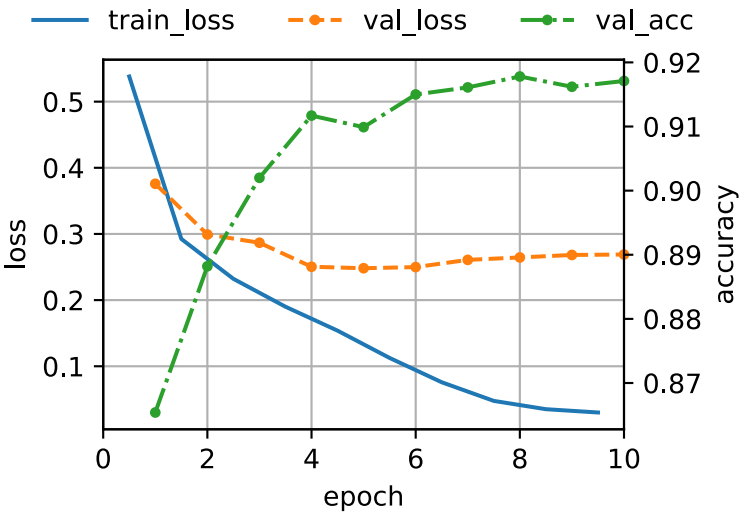
```
macs, macs['VGG-style'] / macs['Mini-MobileNet']
```

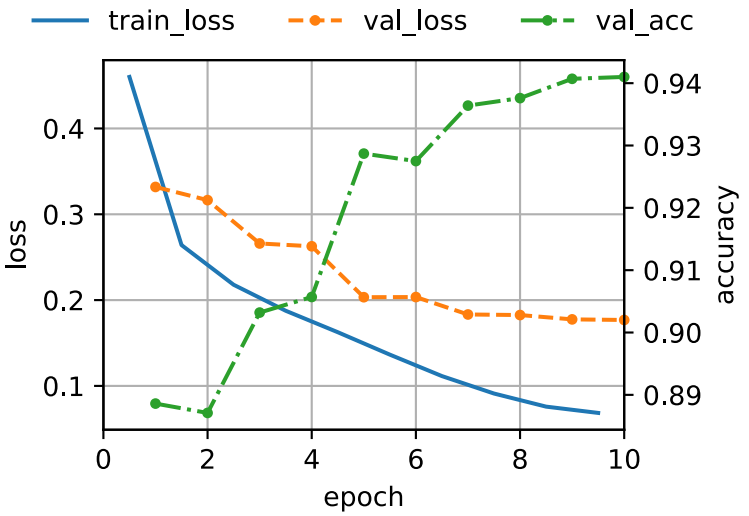
```
({'Mini-MobileNet': 50259968,
  'VGG-style': 384861440,
  'VGG-style, narrow': 49103308},
 7.657415142007253)
```

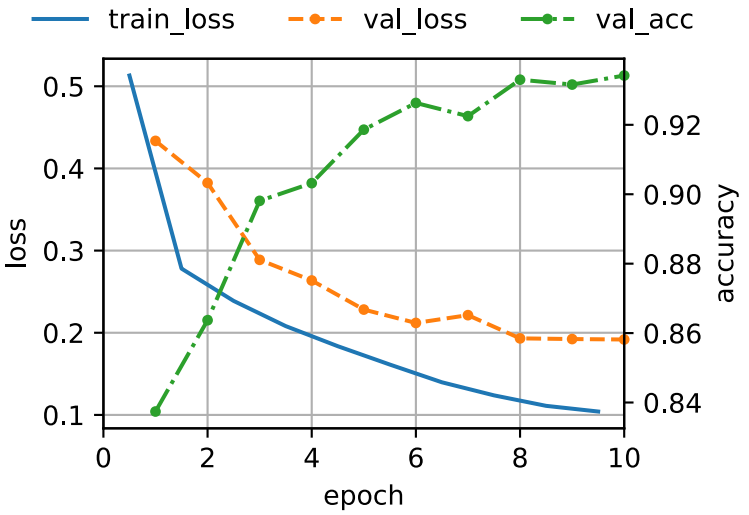
At about the same parameter count, the dense network performs 385 million multiply-adds per image and the separable one 50 million, 7.7 times fewer. The dense network performs most of them in its first three stages, on feature maps of 96 to 24 pixels on a side, where each weight is applied at thousands of positions. MobileNet holds nearly three quarters of its weights in the pointwise convolutions of its last two blocks, which see only 6×6 feature maps.

The comparison that matters on a phone is the one at equal arithmetic. With MobileNet's 50 million multiply-adds, the dense network has to shrink every width by a factor of about 2.8, to 11, 23, 46, and 46 channels (the tuple `narrow` above). That leaves it 76,000 parameters, a seventh of MobileNet's. We train all three networks for 10 epochs on Fashion-MNIST, upsampled to 96×96 pixels as for the networks of the previous sections.

```
data = d2l.FashionMNIST(batch_size=128, resize=(96, 96))
num_steps = 10 * len(data.train_dataloader())  # 10 epochs of 469 steps
models = {'Mini-MobileNet': MiniMobileNet(lr=0.05, num_steps=num_steps),
          'VGG-style': VGGSmall(lr=0.05, num_steps=num_steps),
          'VGG-style, narrow': VGGSmall(narrow, lr=0.05, num_steps=num_steps)}
accuracy = {}
for name, model in models.items():
    trainer = d2l.Trainer(max_epochs=10, num_gpus=1)
    trainer.fit(model, data)
    accuracy[name] = trainer.history['val_acc'][-1]
```







Multiply-adds count arithmetic, not time. To see how they translate into time, we measure how long one forward pass takes on a batch of 128 images, the work of inference, on the GPU. As in Section 5.7.2.1, we run the computation once to warm up, repeat it, and wait for the GPU to finish before reading the clock.

```
def forward_ms(model, num_reps=50):
    """Milliseconds per forward pass of a batch of 128 images."""
    X = torch.randn(128, 1, 96, 96, device=d2l.try_gpu())
    model.eval()
    with torch.inference_mode():
        sync = torch.cuda.synchronize if X.is_cuda else lambda: None
        model(X)
        sync()  # Warm up and wait
        start = time.perf_counter()
```

```
        for _ in range(num_reps):
            model(X)
        sync()
    return 1000 * (time.perf_counter() - start) / num_reps

for name, model in models.items():
    print(f'{name}: {num_params(model):,} parameters, '
          f'{macs[name] / 1e6:.1f}M multiply-adds, '
          f'accuracy {accuracy[name]:.3f}, '
          f'{forward_ms(model):.1f} ms/batch')
```

```
Mini-MobileNet: 542,474 parameters, 50.3M multiply-adds, accuracy 0.917, 2.1 ms/batch
VGG-style: 583,594 parameters, 384.9M multiply-adds, accuracy 0.941, 5.9 ms/batch
VGG-style, narrow: 75,854 parameters, 49.1M multiply-adds, accuracy 0.934, 1.5 ms/batch
```

Table 7.3 collects the results. The accuracies are the range over repeated runs, and the time ratio is the range of each dense network's forward time divided by the mini-MobileNet's.

:Mini-MobileNet against a dense VGG-style network with about the same number of parameters and a narrow one with about the same number of multiply-adds (Fashion-MNIST at 96×96, 10 epochs; validation accuracy and forward-pass time ratio over repeated runs).

Table 7.3: Dws Vs Dense

Model	Parameters	Multiply-adds	Validation accuracy	Forward time
Mini-MobileNet	542,474	50.3 million	91–92%	1
VGG-style	583,594	384.9 million	93.5–94%	2–3.5
VGG-style, narrow	75,854	49.1 million	93%	0.5–0.8

At the same parameter budget the dense network is about two points more accurate, and it performs 7.7 times as much arithmetic to get there. The first half of this sentence should not surprise us. The separable layer is a restricted dense layer (Section 7.5.1.1), and the dense network applies each of its weights at far more positions: it keeps 96×96 feature maps for its first two convolutions, where MobileNet halves the resolution in its stem. Equal parameters are not equal compute.

At equal arithmetic we might expect the comparison to turn around, since this is the budget MobileNet was designed for. It does not. The narrow dense network, with a seventh of MobileNet's parameters, is still about a point more accurate, and its forward pass is faster. On this task, narrowing a dense network is a better way to save arithmetic than factorizing it. Why? By Equation 7.6 the factorization saves the most where the layers are wide: at 11 output channels it saves a factor of 5, at 512 nearly 9. Fashion-MNIST's ten classes of small grayscale images are served well by narrow layers. MobileNet was designed for ImageNet, where its layers are up to 1024 channels wide and a dense network of equal arithmetic would have to be very narrow indeed.

Between MobileNet and the dense network of equal parameters, the time differs by much less than the arithmetic: a factor of 2 to 3.5, where the multiply-adds differ by 7.7. A dense

3×3 convolution with c_o output channels performs $9c_o$ multiply-adds for every input value it reads, enough to keep a GPU's arithmetic units busy. A depthwise convolution performs 9, and a GPU then spends most of the layer's time moving activations to and from memory. Operation counts are only a proxy for time, and how good a proxy depends on the hardware.

7.5.2 Scaling and Searching

MobileNet fixes a block and chooses the channel counts by hand. Two questions remain: how should a network grow when more computation is available, and who should choose the width, depth, and resolution of each stage?

7.5.2.1 Compound Scaling

For the first question it helps to count. The cost of a convnet grows linearly in its depth, quadratically in its width (each layer costs $c_i c_o$), and quadratically in the input resolution (each layer costs hw). Scaling any one of the three alone runs into diminishing returns: a deeper network on small images sees too little detail, and larger images need more layers to cover them with receptive fields. *Compound scaling* (Tan and Le 2019) therefore grows all three together. Each step multiplies the depth by 1.2, the width by 1.1, and the resolution by 1.15, which about doubles the cost, since $1.2 \cdot 1.1^2 \cdot 1.15^2 \approx 1.92$. The number of steps is then the only knob. Three steps, for instance, give a network 1.7 times as deep and 1.3 times as wide, on images 1.5 times as large, at about 7 times the cost.

7.5.2.2 Architecture Search

The second question EfficientNet answered with *neural architecture search*. Its base network was found by an automated search over stacks of inverted-bottleneck blocks, with an objective that rewards accuracy and penalizes multiply-adds. Our measurement above shows the weakness of that objective. EfficientNetV2 (Tan and Le 2021) searched against measured training speed and parameter count instead, and the search replaced the depthwise blocks of the early, high-resolution stages by dense 3×3 convolutions, which are faster there despite their larger operation counts. In both networks the block comes from human insight, and the allocation of width, depth, and resolution to the stages from an optimization.

7.5.2.3 Squeeze-and-Excitation Gates

The inverted bottlenecks that EfficientNet's search stacks carry one more component after the depthwise convolution, a *squeeze-and-excitation* (SE) gate (Hu et al. 2018). A convolution mixes information within a small window. An SE gate lets the network reweight whole channels on the evidence of the entire image. It *squeezes* each channel to one number by global average pooling, passes the resulting vector of c channel summaries through a two-layer MLP with a hidden width of c/r and a sigmoid output (the *excitation*), and multiplies each channel of the input by its gate value in $(0, 1)$. The MLP has about $2c^2/r$ weights for a reduction ratio r and acts on one vector per image, not on the feature map, so it adds almost no computation. It is a simple form of attention, computed per channel rather than per position (Chapter 10 develops the general mechanism).

The gate is small and has become common. MobileNetV3 uses it in many of its blocks, and RegNetY (Section 7.8) adds it to every block of a RegNetX. Our mini-MobileNet does without it, and problem 3 of Section 7.8 adds it to a RegNet. The code is a few lines: pool, two dense layers, rescale.

```
class SE(nn.Module):
    def __init__(self, num_channels, ratio=4):
        super().__init__()
        self.fc = nn.Sequential(
            nn.LazyLinear(num_channels // ratio), nn.ReLU(),
            nn.LazyLinear(num_channels), nn.Sigmoid())

    def forward(self, X):
        s = self.fc(X.mean(dim=(2, 3)))
        return X * s[:, :, None, None]

SE(32)(d2l.randn(2, 32, 16, 16)).shape
```

```
torch.Size([2, 32, 16, 16])
```

For $c = 32$ and $r = 4$ the gate has $2 \cdot 32^2/4 = 512$ weights plus 40 biases. Its output has the shape of its input, so the gate can be inserted into any block without changing the rest of the network.

7.5.3 Structural Re-parameterization

7.5.3.1 The RepVGG Block

ResNet (Section 7.4) showed that networks with several paths per layer train better than plain stacks. At inference time, however, every branch must be computed and its output held in memory until the branches join, and many small operations leave an accelerator underused. A plain stack of 3×3 convolutions maps well to inference hardware: one large, regular operation per layer and one live activation tensor at a time. Can we have both? RepVGG (Ding et al. 2021) shows that we can, by training one network and deploying another. During training, each layer is a three-branch block: a 3×3 convolution, a 1×1 convolution, and an identity path, each followed by its own batch normalization, summed before the ReLU (Figure 7.17). For deployment, the three branches are *fused into a single* 3×3 *convolution* with the same output, because at inference time every branch is linear in its input. The network that is trained has residual paths; the network that is deployed is a plain stack.

```
class RepVGGBlock(nn.Module):
    """Train-time block: 3x3, 1x1, and identity branches, each with BN."""
    def __init__(self, num_channels):
        super().__init__()
        self.conv3 = nn.Conv2d(num_channels, num_channels, kernel_size=3,
                               padding=1, bias=False)
        self.conv1 = nn.Conv2d(num_channels, num_channels, kernel_size=1,
                               bias=False)
        self.bn3, self.bn1, self.bn0 = (nn.BatchNorm2d(num_channels)
                                        for _ in range(3))

    def forward(self, X):
        return F.relu(self.bn3(self.conv3(X)) + self.bn1(self.conv1(X))
                      + self.bn0(X))
```

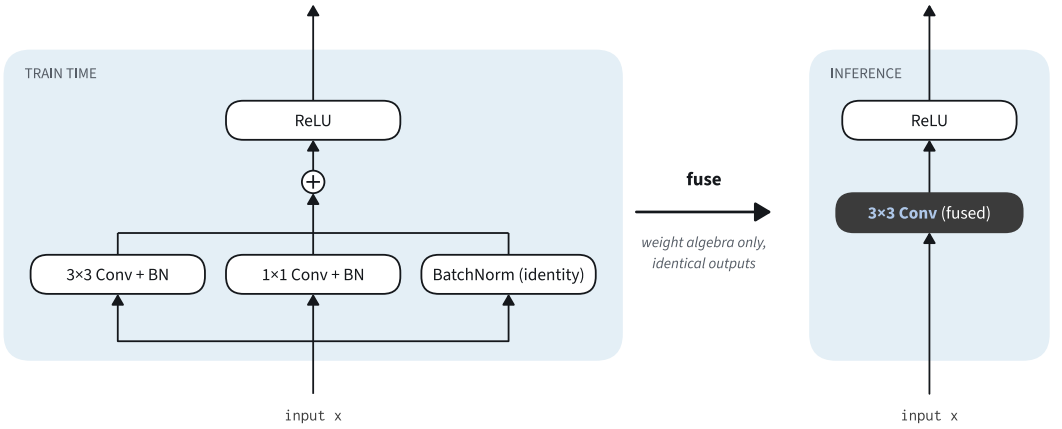


Figure 7.17: Structural re-parameterization. At training time each RepVGG layer sums three branches, a 3×3 convolution, a 1×1 convolution, and an identity, each with batch normalization. For inference the branches are folded into one 3×3 convolution with a bias; the two networks compute the same function.

7.5.3.2 Fusing the Branches

The fusion takes three steps of weight algebra. First, at inference time batch normalization is a fixed affine map of each channel: it subtracts the running mean μ_o, divides by $\sqrt{\sigma_o^2 + \epsilon}$, scales by γ_o, and shifts by β_o. Applied to the output of a convolution with kernel $\mathbf{W}$ and bias b_o, where the index o runs over output channels, it can be absorbed into a new kernel and bias,

$$\hat{\mathbf{W}}_o = \frac{\gamma_o}{\sqrt{\sigma_o^2 + \epsilon}} \mathbf{W}_o, \qquad \hat{b}_o = \beta_o + \frac{\gamma_o (b_o - \mu_o)}{\sqrt{\sigma_o^2 + \epsilon}}. \tag{7.7}$$

The kernel of each output channel is rescaled by that channel's normalization factor, and the shift collects everything that does not depend on the input. For example, $\gamma_o = 2$, $\sigma_o^2 + \epsilon = 16$, $\mu_o = 1$, $\beta_o = 1$, and $b_o = 0$ give $\hat{\mathbf{W}}_o = \mathbf{W}_o/2$ and $\hat{b}_o = 1 - 0.5 = 0.5$. The convolutions of `RepVGGBlock` have no bias, so `fuse_bn` below implements the case $b_o = 0$. Nothing in this step is specific to RepVGG: folding batch normalization into the preceding convolution is a standard optimization for inference.

Second, the two other branches are convolutions in disguise. A 1×1 kernel is a 3×3 kernel that is zero everywhere except at the center, so padding it with a ring of zeros changes nothing. The identity map is a 3×3 convolution whose kernel $\mathbf{W}^{\mathrm{id}}_{o,i}$ is 1 at the center position when $i = o$ and 0 otherwise. Third, convolution is linear in its kernel: three convolutions of the same input, added, equal one convolution with the sum of the three kernels, and the biases add as well. Folding each branch by Equation 7.7, padding the 1×1 kernel, and adding gives the single fused layer (Figure 7.18).

```
def fuse_bn(W, bn):
    """Fold inference-mode batch normalization into a bias-free conv."""
    scale = bn.weight / torch.sqrt(bn.running_var + bn.eps)
```

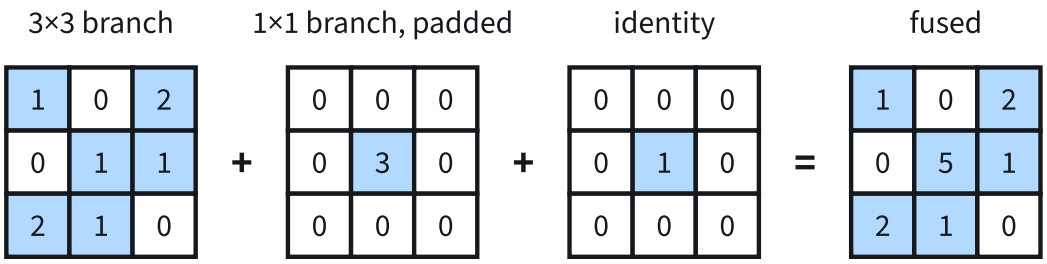


Figure 7.18: The kernels of one output channel o applied to the input channel $i = o$, after batch normalization has been folded into each branch (illustrative values; after folding, the identity's center entry is $\gamma_o/\sqrt{\sigma_o^2 + \epsilon}$ rather than 1). The padded 1×1 kernel and the identity kernel are nonzero only at the center, so the fused kernel differs from the 3×3 branch only there. For $i \neq o$ the identity kernel is zero.

```
    return W * scale.reshape(-1, 1, 1, 1), bn.bias - bn.running_mean * scale

@torch.no_grad()
def fuse_block(blk):
    c = blk.conv3.weight.shape[0]
    W3, b3 = fuse_bn(blk.conv3.weight, blk.bn3)
    W1, b1 = fuse_bn(F.pad(blk.conv1.weight, (1, 1, 1, 1)), blk.bn1)
    I = torch.zeros_like(W3)  # On the device of the block
    I[torch.arange(c), torch.arange(c), 1, 1] = 1.0
    W0, b0 = fuse_bn(I, blk.bn0)
    fused = nn.Conv2d(c, c, kernel_size=3, padding=1, device=W3.device)
    fused.weight.copy_(W3 + W1 + W0)
    fused.bias.copy_(b3 + b1 + b0)
    return fused
```

Fusion uses the *running* statistics of batch normalization, the ones inference uses, so before checking the fused layer we push a few minibatches with mean 1 and standard deviation 2 through the block in training mode, which moves those statistics far from their initial values 0 and 1, and we give the scales γ_o and shifts β_o random values, so that every term of Equation 7.7 matters. The check then feeds fresh data to the block in inference mode and to the fused convolution and reports the largest difference.

```
blk = RepVGGBlock(8)
for _ in range(4):  # Move the BN running statistics
    blk(2 * torch.randn(32, 8, 16, 16) + 1)
with torch.no_grad():  # Nontrivial scales and shifts
    for bn in (blk.bn3, blk.bn1, blk.bn0):
        bn.weight.uniform_(0.5, 1.5)
        bn.bias.normal_()
blk.eval()
fused = fuse_block(blk)
X = 2 * torch.randn(32, 8, 16, 16) + 1
with torch.no_grad():
```

```
    error = (blk(X) - F.relu(fused(X))).abs().max()
float(error)
```

```
4.76837158203125e-06
```

The largest difference is of order 10^{-6}, the roundoff of single-precision arithmetic: the fused convolution computes the same function. It has $9c^2$ weights against the $10c^2$ of the two branch convolutions, a saving of a tenth. The larger change is in how the layer runs: one convolution instead of two convolutions, three normalizations, and two additions, and one live activation tensor instead of three.

7.5.3.3 Quantization after Fusion

Deployment often adds a step that the fusion algebra does not model. To save memory and energy, inference hardware commonly stores weights and activations as 8-bit integers, each tensor represented by 256 evenly spaced values. The fused kernel is a *sum* of branches whose scales, after batch normalization is folded into them, can differ by orders of magnitude, so the fused weights and the activations they produce spread over a wide range, and 256 levels resolve them poorly. The damage can be severe. A fused RepVGG-B0 network quantized naively to 8 bits after training drops from 75.1% to 40.2% top-1 accuracy on ImageNet, where a plain ResNet loses a fraction of a point (Chu et al. 2024). Constraining the branch statistics during training restores the accuracy. Algebraic equivalence in single-precision floating point therefore does not imply equivalence after quantization.

7.5.4 Summary

Efficient convnets treat cost as a design objective alongside accuracy. Depthwise-separable convolutions cut the arithmetic of a layer to $1/c_\mathrm{o} + 1/k^2$ of a dense convolution, and the inverted bottleneck arranges them so that the tensors kept between blocks are the narrow ones. On Fashion-MNIST, a mini-MobileNet comes about two points short of a dense network with the same number of parameters while performing 7.7 times fewer multiply-adds, but its forward pass is only 2 to 3.5 times faster, because depthwise convolutions perform little arithmetic per value they read from memory. A dense network narrowed to the same arithmetic is more accurate than MobileNet and faster still: on a task this small, narrow dense layers are the cheaper choice. Compound scaling and architecture search allocate depth, width, and resolution once the block is fixed, and searches that optimize measured latency rather than operation counts avoid depthwise convolutions where memory traffic dominates. A squeeze-and-excitation gate lets a block rescale its channels on the evidence of the whole image, for a negligible share of the arithmetic. Structural re-parameterization trains a multi-branch network and deploys an equivalent plain stack, with the caveat that equivalence in floating point need not survive quantization. Each of these choices was made by hand or by a search for one network at a time; the last section of this chapter asks how to explore whole families of networks systematically.

7.5.5 Exercises

1. **Channel permutation between groups.** ▣ Grouped layers can also be mixed without dense weights. Follow a grouped $k \times k$ convolution from c_i to c_o channels with g groups by a second grouped convolution from c_o to c_o channels with g groups, and insert a fixed permutation of the c_o intermediate channels between the two layers.

Find the condition on g and c_o under which some permutation makes each output of the second layer depend on all c_i input channels, and construct such a permutation when the condition holds.

2. code **Width multiplier.** Add a width multiplier m to `MiniMobileNet` that scales every channel count (Howard et al. 2017). Predict how the parameter count and the number of multiply-adds scale with m, and verify your prediction at $m = 0.5$ with `num_params` and `mobilenet_macs`.

3. code **Folding batch normalization.** The function `fuse_bn` folds batch normalization into a preceding convolution without a bias, using Equation 7.7 with $b_o = 0$. Extend `fuse_bn` to a convolution with bias b_o, and verify numerically that the convolution followed by batch normalization in inference mode agrees with the folded convolution. Then switch batch normalization to training mode, feed the same example inside two different minibatches, and explain why no fixed kernel and bias reproduce both outputs.

4. code **The inverted bottleneck.** ▣ MobileNetV2 (Sandler et al. 2018) replaces the block of Figure 7.15 by an inverted bottleneck: a 1×1 expansion by a factor t, a depthwise 3×3 convolution, a linear 1×1 projection, and a residual connection whenever input and output shapes match.

    1. Implement the block with $t = 6$ and replace the blocks of `MiniMobileNet` by it, choosing the widths so that the parameter count stays near 542,474. Train it with the setup of Table 7.3, and compare its validation accuracy and its forward time with those of `MiniMobileNet` on the same device.
    2. Add a ReLU after the projection. Train the linear and the ReLU variants three times each, and compare the gap in validation accuracy with the spread across runs. Does the paper's ablation, which favors the linear projection, reproduce at this scale?
    3. For a 96×96 input, tabulate for each block of both networks the number of elements in its input, its output, and its largest internal tensor. Which network needs more activation memory when every tensor is held in full? Show that an inference implementation of the inverted bottleneck never needs to hold a whole expanded tensor, and determine which tensors then set its peak activation memory.

5. code **Width versus resolution.** ▣ Lowering the input resolution is a second way to cut multiply-adds. Using the width multiplier of problem 2, train `MiniMobileNet` at $m = 1$ on 48×48 inputs and at $m = 0.5$ on 96×96 inputs, twice each and otherwise with the setup of Table 7.3. Compare the two networks in multiply-adds, parameter count, and validation accuracy. Fashion-MNIST images have 28×28 pixels. How does this limit what the comparison says about scaling resolution in EfficientNet (Tan and Le 2019)?

6. code **Arithmetic intensity.** ▣ `MiniMobileNet` performs 7.7 times fewer multiply-adds than `VGGSmall`, yet its forward pass in Table 7.3 is only 2 to 3.5 times faster.

    1. The arithmetic intensity of a layer is its number of multiply-adds divided by the number of values it reads and writes: weights, input activations, and output activations. For every block of `MiniMobileNet`, compute it for the depthwise convolution, for the pointwise convolution, and for a dense 3×3 convolution with the block's input width, output width, and stride. In which stages is the ratio between

the dense and the depthwise intensity largest? Compare them with the stages in which EfficientNetV2 (Tan and Le 2021) replaces depthwise blocks by dense ones.
2. Measure `forward_ms` for both networks at batch sizes 1, 16, 128, and 512, and plot the time per image against the batch size. How does the ratio of the two networks' times change with the batch size, and why?

7. code **Normalization before the convolution.** ▯ Let batch normalization come before a 3×3 convolution with zero padding of 1, with no nonlinearity in between. Fold it into the kernel and bias of the convolution and compare the folded layer with the original numerically. At which output positions do they disagree, and with what value would the input have to be padded for the fold to be exact?

8. code **Training with RepVGG blocks.** ▯ If the fused RepVGG layer computes the same function, why not train the plain 3×3 stack directly?

    1. Give `RepVGGBlock` a stride and an output width. A downsampling block has stride 2 and a new width, so an identity branch cannot match its shape. Such a block keeps only its 3×3 and 1×1 branches (Ding et al. 2021). Build a small network from these blocks and a plain network that replaces each block by one 3×3 convolution of the same width and stride, followed by batch normalization and a ReLU. Train both on Fashion-MNIST and compare the training curves.
    2. Extend `fuse_block` to the downsampling blocks, fuse the trained RepVGG network, and confirm that its validation accuracy is unchanged.
    3. Fold the batch normalization of the plain network into its convolutions with `fuse_bn`. Then quantize the weights of both deployed networks to eight bits with a symmetric per-tensor scale, $\hat{w} = s \cdot \mathrm{round}(w/s)$ with $s = \max|w|/127$, and compare the accuracy drops. Relate the outcome to the distribution of the fused weights, as Chu et al. (2024) analyze.

7.6 Modern Training Recipes for Convnets

So far in this chapter we have tried to hold the training procedure fixed within each section, so that differences in accuracy could be credited to the architectures. Even so, the peak learning rate alone moved VGG by a point and a half (Section 7.2). Published comparisons are often made with far less care: a new architecture is trained with the latest methods and compared with a baseline trained years earlier. How much of a reported accuracy comes from the *training recipe* (the optimizer, schedule, augmentation, regularization, and length of training) rather than from the network itself?

ResNet-50, the deeper relative of the ResNet-18 of Section 7.4, gives a clear answer. Wightman et al. (2021) changed nothing about its architecture and trained it with the methods in use around 2022 instead of those of its 2015 debut. Its ImageNet top-1 accuracy rose from 76.1% to 80.4%. That gain, from the recipe alone, is as large as the gains that many new architectures of those years reported over ResNet-50. So when a paper reports two points over a baseline trained with an older recipe, we cannot tell whether the architecture or the recipe deserves the credit.

This section lists what changed between 2015 and 2022, implements each ingredient in a few lines, and then trains one ResNet-18 under an old and a new recipe to measure the difference. None of the ingredients is specific to convnets. The same ones are used to train transformers, so what follows is general training technique.

```
import random
```

7.6.1 Training Recipes from 2015 to 2022

The original ResNet recipe was already careful: stochastic gradient descent with momentum (Section 7.5), weight decay, a learning rate divided by 10 twice during training, and random crops and horizontal flips as augmentation (He et al. 2016). Over the following years every one of these choices was revisited, first one technique at a time and then in studies that combined them (He et al. 2019; Wightman et al. 2021). Table 7.4 summarizes the shift. Do not worry about the names in it: we explain each ingredient where we implement it below.

:A 2015 ImageNet recipe and a representative high-accuracy recipe from the early 2020s. The later column is not a universal standard; strong recipes still vary by model and compute budget.

Table 7.4: Recipe 2015 2022

Ingredient	2015 recipe	2022 recipe
Optimizer	SGD with momentum	AdamW or LAMB; tuned SGD remains competitive
Schedule	step decay (divide by 10 twice)	cosine decay with linear warmup
Augmentation	random crops, horizontal flips	RandAugment, Mixup, CutMix, random erasing
Regularization	weight decay	+ label smoothing, stochastic depth
Duration	90 epochs	300 to 600 epochs
Evaluated weights	final weights	exponential moving average of the weights

Two remarks help to read the table. First, its rows are not independent. Every augmentation and every regularizer makes the training task harder, so the network needs more epochs to fit it, and the *duration* row grows from 90 epochs to several hundred. Regularization and duration have to grow together, since a heavily regularized short run can do worse than a lightly regularized one.

Second, the *optimizer* row records a common change, not a universal one. Adam, which Chapter 9 treats in full, divides each coordinate of the gradient by a running estimate of its typical size, so that every weight moves at about the pace that the learning rate sets. Earlier implementations added weight decay to the gradient, where this rescaling distorts it. AdamW (Loshchilov and Hutter 2019) applies the decay to the weights directly: each step subtracts from every weight the fraction (learning rate times decay) of its value. This made adaptive methods competitive with tuned SGD on many vision tasks. (LAMB is a relative of Adam for very large minibatches; we will not need it.)

7.6.2 Implementing the Ingredients

Each ingredient takes a few lines of code. We implement label smoothing, the cosine schedule, Mixup, stochastic depth, and the weight average. RandAugment (Cubuk et al. 2020), which applies randomly chosen image transformations at one tuned strength, is a library call in practice. Its contribution lies in the tuning and not in the code, so we omit it.

7.6.2.1 Label Smoothing

With one-hot targets, the cross-entropy loss of a correctly classified example keeps decreasing as the logit of the correct class grows, so the loss can never reach its minimum with finite logits and the network is rewarded for ever larger confidence. Label smoothing (Szegedy et al. 2016) replaces the one-hot target by a mixture that puts weight $1 - \epsilon$ on the label and spreads ϵ uniformly over all K classes. Writing p_k for the predicted softmax probabilities, the loss on an example with label y becomes

$$l_\epsilon = -(1-\epsilon)\log p_y - \frac{\epsilon}{K}\sum_{k=1}^{K}\log p_k. \tag{7.8}$$

The first term is the ordinary cross-entropy, weighted by $1-\epsilon$; the second penalizes any class whose probability approaches zero. The loss is minimized by the probabilities $1-\epsilon+\epsilon/K$ on the correct class and ϵ/K on every other class, which a finite gap between the logits attains. For a prediction that is already very confident and correct, the unsmoothed loss is nearly zero, while the smoothed loss stays well above zero:

```
logits, y = d2l.tensor([[10.0, 0.0, 0.0]]), d2l.tensor([0])
for epsilon in (0.0, 0.1):
    loss = F.cross_entropy(logits, y, label_smoothing=epsilon)
    print(f'epsilon={epsilon}: loss={float(loss):.4f}')
```

```
epsilon=0.0: loss=0.0001
epsilon=0.1: loss=0.6668
```

With $\epsilon = 0.1$ the loss of this example is 0.67 instead of 0.0001. Is 0.67 large? With three classes the smoothed loss cannot fall below about 0.29, a value it reaches when the correct logit leads the others by about 3.3. A lead of 10 is therefore penalized as overconfident, and the gradient now pulls the logits back together instead of pushing them further apart.

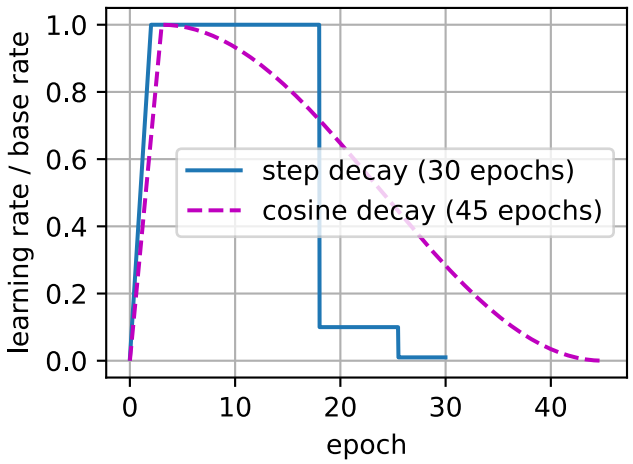
7.6.2.2 Cosine Schedules with Warmup

A step schedule holds the learning rate constant for tens of epochs and then divides it by 10, an abrupt change whose two times are extra hyperparameters. The modern default (Loshchilov and Hutter 2016) is already familiar. It is the schedule with which we have trained the networks of this chapter since AlexNet (Section 7.1): a linear *warmup* from zero to the base rate η_0 over the first t_w steps, then the half-cosine *decay* Equation 7.1 to zero at the last step T. To the reasons given there we add one. During the first small steps, the running statistics of batch normalization and of the optimizer can settle before full-size updates arrive. The cosine also has no drop times to tune. Recall that `d2l.cosine_warmup(t, num_steps, warmup_steps)` returns the factor by which the schedule multiplies the base rate at step t, and that the optimizer updates the rate after every minibatch. We plot it next to the

step decay of the 2015 recipe. Each schedule runs over the budget it will get in the experiment at the end of this section: 30 epochs for the step decay, preceded there by a two-epoch warmup, and 45 epochs for the cosine, at 79 minibatches per epoch.

```
def step_decay(t, num_steps, warmup_steps):
    """Recipe A: linear warmup, then divide by 10 at 60% and 85% of steps."""
    warmup = min((t + 1) / warmup_steps, 1)
    return warmup * 0.1 ** sum(t >= int(f * num_steps) for f in (0.6, 0.85))

steps_per_epoch = 79
steps_a = list(range(30 * steps_per_epoch))  # Recipe A: 30 epochs
steps_b = list(range(45 * steps_per_epoch))  # Recipe B: 45 epochs
d2l.plot([[t / steps_per_epoch for t in steps_a],
          [t / steps_per_epoch for t in steps_b]],
         [[step_decay(t, len(steps_a), 2 * steps_per_epoch) for t in steps_a],
          [float(d2l.cosine_warmup(t, len(steps_b), 3 * steps_per_epoch))
           for t in steps_b]],
         xlabel='epoch', ylabel='learning rate / base rate',
         legend=['step decay (30 epochs)', 'cosine decay (45 epochs)'])
```



The cell below builds recipe B's schedule for a base rate of 0.002 and prints the rate at the first step, at the end of warmup, halfway through the decay, and at the last step.

`LambdaLR` multiplies the base rate of the optimizer by a function of the step. Calling `scheduler.step()` after every optimizer step advances it by one minibatch.

```
T, t_w = 45 * 79, 3 * 79
optim = torch.optim.SGD([nn.Parameter(torch.zeros(1))], lr=2e-3)
scheduler = torch.optim.lr_scheduler.LambdaLR(
    optim, lambda t: d2l.cosine_warmup(t, T, t_w))
rates = []
for t in range(T):  # One optimizer step and one scheduler step per minibatch
    rates.append(scheduler.get_last_lr()[0])
    optim.step()
    scheduler.step()
```

```
[round(rates[t], 6) for t in (0, t_w, (T + t_w) // 2, T - 1)]
```

```
[0.0, 0.002, 0.001, 0.0]
```

The rates are 0, 0.002, 0.001, and nearly 0, as Equation 7.1 predicts: nothing, the base rate, half of it, and nothing again.

7.6.2.3 Mixup

Mixup (Zhang et al. 2018) trains on convex combinations of pairs of examples. For each minibatch it draws a mixing weight $\lambda \sim \text{Beta}(\alpha, \alpha)$ and a random partner j for every example i, and trains on

$$\tilde{\mathbf{x}} = \lambda \mathbf{x}_i + (1-\lambda)\mathbf{x}_j, \qquad \tilde{\mathbf{y}} = \lambda \mathbf{y}_i + (1-\lambda)\mathbf{y}_j, \tag{7.9}$$

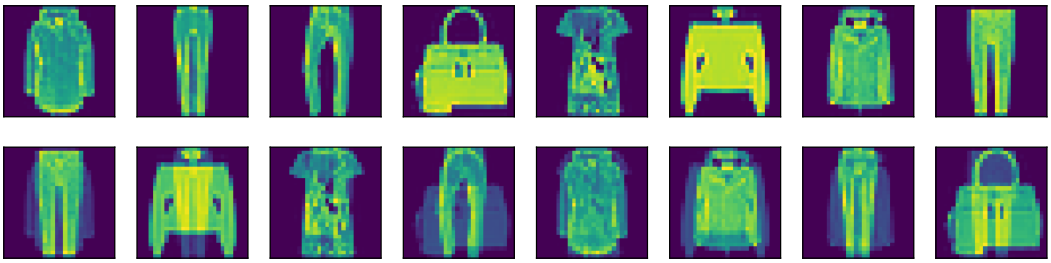
where $\mathbf{y}_i$ and $\mathbf{y}_j$ are one-hot label vectors, so the mixed label $\tilde{\mathbf{y}}$ is a distribution over classes. The cross-entropy is linear in the target distribution, so the loss on $\tilde{\mathbf{y}}$ equals the λ-weighted sum of the losses on the two original labels, and we implement it that way. Mixup asks the network to behave linearly between training examples, which discourages it from memorizing individual examples. With the usual small α (0.1 to 0.2), the Beta distribution puts most of its mass near 0 and 1, so most minibatches are barely mixed and a few are blended strongly. The function below mixes a minibatch with one λ for all of it. We save it in the `d2l` package because the ConvNeXt section reuses it.

```
def mixup(X, y, alpha):
    """Mix a minibatch with a shuffled copy; return both label sets."""
    lam = random.betavariate(alpha, alpha)
    perm = torch.randperm(X.shape[0], device=X.device)
    return lam * X + (1 - lam) * X[perm], y, y[perm], lam
```

To see the blending, we mix a minibatch of Fashion-MNIST images with $\alpha = 2$, which favors values of λ near the middle of its range (training uses a much smaller α). The top row shows the originals, the bottom row their mixtures; the printed λ is drawn afresh in every run.

```
data = d2l.FashionMNIST(batch_size=8)
X, y = next(iter(data.train_dataloader()))
X_mix, y_a, y_b, lam = mixup(X, y, alpha=2.0)
d2l.show_images(torch.cat([X, X_mix]).squeeze(1), 2, 8)
print(f'lambda = {lam:.2f}')
```

```
lambda = 0.27
```



A mixed image is a double exposure that no camera would take. CutMix (Yun et al. 2019) mixes in space instead: it cuts a rectangle from one image and pastes it into the other, and mixes the labels in proportion to the area. Every pixel then comes from a real image. Modern recipes alternate randomly between Mixup and CutMix from one minibatch to the next. The exercises implement it.

7.6.2.4 Stochastic Depth

Dropout (Section 4.6) sets randomly chosen activations to zero during training. Stochastic depth (Huang et al. 2016) applies the same idea to whole residual branches: during training, the branch of each residual block is dropped with probability p, so the block passes its input through unchanged and the step trains a shallower network. Residual connections make this safe, since the identity path always carries the signal. For a block that computes $\mathbf{x}+f(\mathbf{x})$, the branch becomes

$$\mathbf{x} + \frac{b}{1-p} f(\mathbf{x}), \qquad b \sim \text{Bernoulli}(1-p). \tag{7.10}$$

The expected value of $b/(1-p)$ is 1, so the expected output of the branch is unchanged, and at evaluation time the block runs without dropping, as with inverted dropout. Modern implementations draw b separately for every example in the minibatch. The function `drop_path` does this, and we save it for the ConvNeXt section. The block below subclasses the `Residual` block of Section 7.4 (in the residual block the final ReLU follows the addition) and drops its branch before the addition. Running 1,000 examples through it in training mode, the fraction of examples whose output equals the ReLU of the input, the ones whose branch was dropped, should be close to p.

```
def drop_path(Y, p, training):
    """Zero each example of Y with probability p, rescale the rest."""
    if not training or p == 0:
        return Y
    shape = (Y.shape[0],) + (1,) * (Y.ndim - 1)  # One draw per example
    keep = torch.rand(shape, device=Y.device) > p
    return Y * keep / (1 - p)

class StochasticResidual(d2l.Residual):
    """A residual block whose branch is dropped with probability p."""
    def __init__(self, num_channels, p):
        super().__init__(num_channels)
        self.p = p
```

```
    def forward(self, X):
        Y = self.bn2(self.conv2(F.relu(self.bn1(self.conv1(X)))))
        return F.relu(X + drop_path(Y, self.p, self.training))

blk, X = StochasticResidual(3, p=0.5), torch.randn(1000, 3, 8, 8)
Y = blk(X)  # A module is in training mode until .eval() is called
dropped = (Y == F.relu(X)).flatten(1).all(1).float().mean()
print(f'fraction of examples with a dropped branch: {float(dropped):.3f}')
```

```
fraction of examples with a dropped branch: 0.503
```

In deep networks the drop probability usually grows linearly from 0 in the first block to a maximum between 0.1 and 0.5 in the last. The per-example implementation computes every branch before masking it, so it regularizes but saves no arithmetic. The original method draws one b per block and minibatch and can skip a dropped branch entirely, which shortens training.

7.6.2.5 Averaging Weights

The last row of Table 7.4 costs almost nothing. Late in training, stochastic gradients make the parameters fluctuate around a region of low loss, and an average of recent parameters lies closer to the middle of that region than any single one. Stochastic weight averaging (Izmailov et al. 2018) takes an equal-weight average of iterates late in training. Modern recipes more often keep an exponential average, of the kind that batch normalization keeps of its statistics (Section 7.3): a *shadow copy* $\bar{\theta}$ of the parameters into which, after every update, the current ones are blended with a decay ρ:

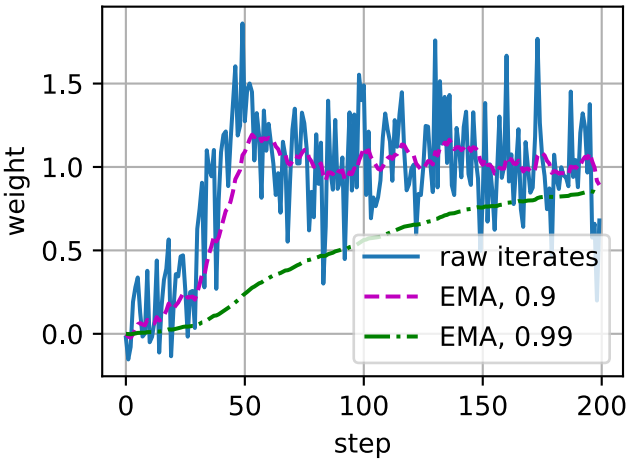
$$\bar{\theta} \leftarrow \rho\bar{\theta} + (1-\rho)\theta. \tag{7.11}$$

This is an *exponential moving average* (EMA): the parameters of k steps ago enter $\bar{\theta}$ with weight $(1-\rho)\rho^k$, so the average reaches back about $1/(1-\rho)$ steps, 100 steps for $\rho = 0.99$. Training updates θ as usual; only evaluation uses $\bar{\theta}$. The class below is the whole mechanism. It averages the trainable parameters; the running statistics of batch normalization are not parameters, and the exercises ask which statistics the averaged network should use. We apply it, with two decays, to a one-weight model whose weight we set by hand, as training might move it: it starts at 0, rises to 1 over the first 50 steps, and then fluctuates around 1, as stochastic gradient steps would.

```
class EMA:
    """A shadow copy of a model's parameters, updated after each step."""
    def __init__(self, model, decay=0.99):
        self.decay = decay
        self.shadow = {k: p.detach().clone()
                       for k, p in model.named_parameters()}

    @torch.no_grad()
    def update(self, model):
        for k, p in model.named_parameters():
            self.shadow[k].lerp_(p, 1 - self.decay)  # beta*avg + (1-beta)*p
```

```
net = nn.Linear(1, 1, bias=False)
nn.init.zeros_(net.weight)  # Training starts at 0
emas = {decay: EMA(net, decay) for decay in (0.9, 0.99)}
raw, avg = [], {decay: [] for decay in emas}
for t in range(200):  # Iterates rise to 1 over 50 steps, then stay noisy
    with torch.no_grad():
        net.weight.copy_(min(t / 50, 1) + 0.3 * torch.randn(1, 1))
    raw.append(float(net.weight.detach()))
    for decay, ema in emas.items():
        ema.update(net)
        avg[decay].append(float(ema.shadow['weight']))
d2l.plot(list(range(200)), [raw, *avg.values()], xlabel='step',
         ylabel='weight', legend=['raw iterates', 'EMA, 0.9', 'EMA, 0.99'])
```



With $\rho = 0.9$ the average reaches back about 10 steps. It follows the rise about 10 steps late and then fluctuates far less than the iterates around their mean. With $\rho = 0.99$ it reaches back about 100 steps: it is smoother still, but it lags far behind the rise and is still about a fifth short of 1 when the run ends. This is the risk of the method. If $1/(1-\rho)$ steps is long compared with the time over which the weights are still improving, the average lags behind them and evaluates worse than the final weights. Recipes therefore choose ρ close to 1 (0.999 or 0.9999) only for long runs.

7.6.3 Two Recipes on One Network

We now combine the ingredients and measure what they add. The network is the ResNet-18 of Section 7.4, rebuilt from the library's `Residual` block, with the He initialization of Section 4.4 for all convolution and linear weights. The task is Fashion-MNIST at 96×96, with one change: we train on a random subset of 10,000 of the 60,000 training images. With less data, regularization matters more, and the experiment stays small enough to rerun. Each run draws its own subset, and both recipes of a run share it. Each recipe takes the base rate `lr` and the total number of optimizer steps `num_steps`, from which its schedule is computed.

```
class FashionMNIST10k(d2l.FashionMNIST):
    """Fashion-MNIST with a random subset of num_train training images."""
    def __init__(self, batch_size=128, resize=(96, 96), num_train=10000):
        super().__init__(batch_size, resize)
        idx = torch.randperm(len(self.train))[:num_train]
        self.train = torch.utils.data.Subset(self.train, idx)

class ResNet18(d2l.Classifier):
    """The ResNet-18 of the residual-networks section, from d2l.Residual."""
    def __init__(self, lr, num_steps, num_classes=10):
        super().__init__()
        self.save_hyperparameters()
        layers = [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)]
        for i, c in enumerate((64, 128, 256, 512)):
            for j in range(2):
                down = i > 0 and j == 0
                layers.append(d2l.Residual(c, use_1x1conv=down,
                                           strides=2 if down else 1))
        layers += [nn.AdaptiveAvgPool2d((1, 1)), nn.Flatten(),
                   nn.LazyLinear(num_classes)]
        self.net = nn.Sequential(*layers)

    def weight_init(self, module):  # The Trainer applies it after a dry run
        if isinstance(module, (nn.Conv2d, nn.Linear)):
            nn.init.kaiming_normal_(module.weight)  # Variance 2/n_in
            if module.bias is not None:
                nn.init.zeros_(module.bias)
```

The network and the data are the same for both recipes. Almost everything else differs: the optimizer, the weight decay, the schedule, the training loss, and the length of training. *Recipe A* follows 2015: SGD with momentum 0.9, weight decay 5×10^{-4} added to the gradient, a base rate of 0.05 reached by a linear warmup over the first two epochs and divided by 10 at 60% and at 85% of the steps, the plain cross-entropy loss, and 30 epochs. The warmup is our one concession to later practice. Without it, this rate makes the loss diverge in the first epoch in about a third of the runs, and a short warmup is the standard remedy for large rates with batch normalization (Goyal et al. 2017). The *compact modern recipe* B uses AdamW with weight decay 0.05, a base rate of 0.002 with three epochs of warmup and a cosine decay, label smoothing with $\epsilon = 0.1$, Mixup with $\alpha = 0.2$, and 45 epochs, since longer training is part of the recipe. It omits RandAugment, CutMix, stochastic depth, and weight averaging to keep the experiment small, so its result measures this subset of the full recipe.

The two weight decays, 0.05 and 5×10^{-4}, look a hundredfold apart. They are closer than that. What shrinks a weight in one step is the learning rate times the decay. At the peak rate this is $0.002 \times 0.05 = 10^{-4}$ of its value for AdamW and $0.05 \times 5 \times 10^{-4} = 2.5 \times 10^{-5}$ for SGD, which momentum multiplies by up to 10. Of the two, SGD's decay is then the stronger one. Finally, both recipes report the plain cross-entropy as validation loss; label smoothing and Mixup change only the training loss.

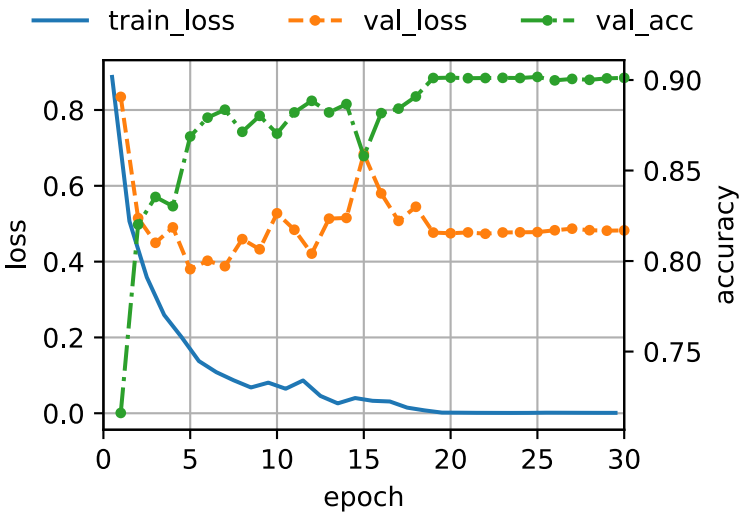
```
class ClassicRecipe(ResNet18):
    """Recipe A: SGD with momentum, weight decay, warmup, step decay."""
    def configure_optimizers(self):
        optim = torch.optim.SGD(self.parameters(), lr=self.lr, momentum=0.9,
                                weight_decay=5e-4)
        warmup = 2 * self.num_steps // 30  # Two of the 30 epochs
        return optim, torch.optim.lr_scheduler.LambdaLR(
            optim, lambda t: step_decay(t, self.num_steps, warmup))

class ModernRecipe(ResNet18):
    """Compact modern recipe: AdamW, warmup and cosine, smoothing, Mixup."""
    def configure_optimizers(self):
        warmup = 3 * self.num_steps // 45  # Three of the 45 epochs
        optim = torch.optim.AdamW(self.parameters(), lr=self.lr,
                                  weight_decay=0.05)
        return optim, torch.optim.lr_scheduler.LambdaLR(
            optim, lambda t: d2l.cosine_warmup(t, self.num_steps, warmup))

    def training_step(self, batch):
        X, y_a, y_b, lam = mixup(*batch, alpha=0.2)
        Y_hat = self(X)
        smooth = lambda y: F.cross_entropy(Y_hat, y, label_smoothing=0.1)
        return lam * smooth(y_a) + (1 - lam) * smooth(y_b)
```

We first train recipe A for 30 epochs and evaluate on the 10,000 validation images, which no run trains on.

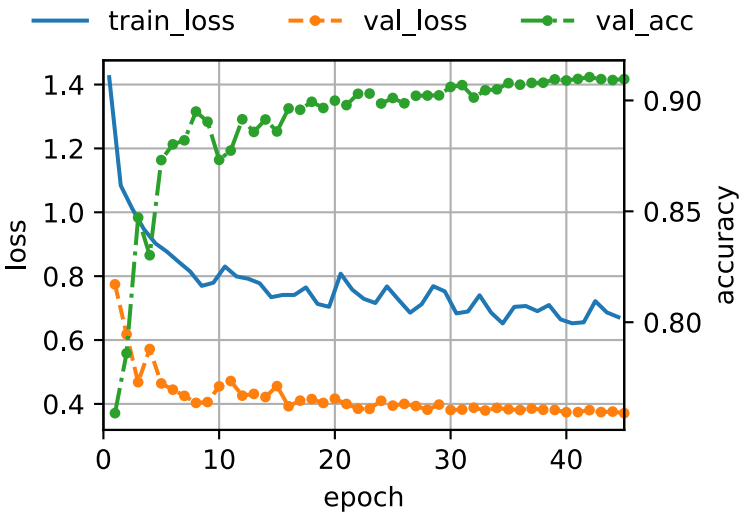
```
data = FashionMNIST10k()
steps_per_epoch = len(data.train_dataloader())  # 79 minibatches of <= 128
model = ClassicRecipe(lr=0.05, num_steps=30 * steps_per_epoch)
trainer = d2l.Trainer(max_epochs=30, num_gpus=1)
trainer.fit(model, data)
trainer.history['val_acc'][-1]
```




```
0.9012
```

Then the compact modern recipe trains the same network on the same images for 45 epochs.

```
model = ModernRecipe(lr=2e-3, num_steps=45 * steps_per_epoch)
trainer = d2l.Trainer(max_epochs=45, num_gpus=1)
trainer.fit(model, data)
trainer.history['val_acc'][-1]
```




```
0.9096
```

Table 7.5 collects the validation accuracy over repeated runs, each with its own subset, initialization, and minibatch order.

:The same ResNet-18 under both recipes (10,000 Fashion-MNIST training images at 96×96; validation accuracy, range over repeated runs).

Table 7.5: Recipe Results

Recipe	Epochs	Validation accuracy
A (2015): SGD, warmup, step decay	30	89–90.5%
Compact modern: AdamW, cosine, smoothing, Mixup	45	90.5–91%

The gain is small but consistent. In every pair of our repeated runs, the pair above included, the compact modern recipe beats recipe A trained on the same subset, by about half a point to a point and a half. Rerun the two cells to draw a pair of your own. On ImageNet the full recipe adds four points; with 10,000 small grayscale images, a 45-epoch budget, and only part of the menu, the effect is a fraction of that. Note, however, what the experiment cannot tell us. Which ingredient produced the point: Mixup, the optimizer, or simply 15 more epochs? We changed all of them at once. Answering takes an ablation, which changes one ingredient at a time, and we leave it to the exercises.

The training curves are instructive in their own right. Under recipe A the training loss falls to zero, while the validation loss ends between 0.4 and 0.5, above the minimum it reached early in training, even though the accuracy kept improving. This is the growing overconfidence that label smoothing was introduced to prevent. Under the modern recipe the validation loss keeps falling and ends below 0.4. Its training loss, however, stays far above recipe A's and even above its own validation loss, because label smoothing and Mixup make the training targets harder to fit, while the validation loss is the plain cross-entropy on unmixed validation images. Training losses can therefore not be compared across recipes. Nor do hyperparameters transfer between them: AdamW's base rate is 25 times smaller than SGD's, so an ablation that swaps one ingredient must retune the others around it. This is why careful recipe ablations, such as those of Wightman et al. (2021), are expensive.

7.6.4 Comparing Published Results

How, then, should a table of published accuracies be read? With some suspicion. The recipe alone moves a fixed ResNet-50 by four points, so a gain of a point on ImageNet between two architectures says little unless both were trained with comparable recipes. Careful papers therefore retrain their baselines with their own recipe, and tables that collect numbers from papers of different years do not support conclusions drawn from one-point gaps.

The test set adds a second source of uncertainty once accuracies are close. On ImageNet-V2, a new test set collected by following the original protocol (Deng et al. 2009), models lost 11 to 14 points, yet their ranking was preserved (Recht et al. 2019). The images were slightly harder; the models had not overfitted to the old test set. A relabeling of the original validation set with several labels per image found that many apparent errors are ambiguous images or images with several objects (Beyer et al. 2020), so one-point gains measured on the original labels need not hold under better ones. For networks whose top-1 accuracies lie within a point, differences on other tasks, such as locating objects in an image, are more informative.

7.6.5 Summary

Between 2015 and the early 2020s, high-accuracy recipes for convnets changed along every axis: optimizer, schedule, augmentation, regularization, length of training, and the weights that are evaluated. On a fixed ResNet-50 the changes add about four ImageNet points. Each ingredient takes a few lines: a smoothed target, a schedule function, a convex combination of two minibatches, a random mask on a residual branch, and a shadow copy of the parameters. On a ResNet-18 trained on 10,000 Fashion-MNIST images, a compact subset of the menu with 15 more epochs gains half a point to a point and a half over the 2015 recipe, and the experiment does not say which ingredient is responsible. Architecture comparisons therefore need matched training procedures, and with a fixed network and budget, tuning the procedure can be cheaper than changing the architecture. Section 7.7 starts from the modern recipe and asks how far a convnet's architecture itself can be modernized.

7.6.6 Exercises

1. **Binary cross-entropy and Mixup.** Wightman et al. (2021) train with a binary cross-entropy loss that treats each class as an independent yes/no target. Explain why this loss pairs naturally with Mixup and CutMix. Consider an image that contains parts of two classes, and contrast how softmax and sigmoid outputs can represent it.

2. code **CutMix.** CutMix (Yun et al. 2019) draws λ uniformly from $[0, 1]$, that is, from Beta$(1, 1)$ rather than the Beta$(0.2, 0.2)$ the compact modern recipe uses for Mixup. It pastes into each image a rectangle from its shuffled partner that covers a fraction $1 - \lambda$ of the image area. Implement CutMix as a variant of `mixup`, and mix the labels as in Equation 7.9 with λ replaced by the fraction of the area actually kept, since a rectangle clipped at the image border covers less than $1 - \lambda$. Train the compact modern recipe on `FashionMNIST10k` once with CutMix in place of Mixup and once with the two alternating at random from minibatch to minibatch, as modern recipes do. Compare the validation accuracy of both runs with the range in Table 7.5.

3. code **Weight averaging horizon.** Add the `EMA` class to `ModernRecipe`: update the shadow copy after every optimizer step, by subclassing `d2l.Trainer`, and evaluate with the shadow weights. Sweep the decay ρ of Equation 7.11 over 0.9, 0.99, and 0.999. Count the optimizer steps at which `d2l.cosine_warmup` holds the learning rate below a tenth of its peak, and relate the best decay to that count through the averaging horizon of about $1/(1-\rho)$ steps. Batch normalization keeps running statistics that are not parameters; which statistics should the averaged model use?

4. code **Recipe ablation.** ⊠ In Table 7.5 the compact modern recipe beats recipe A by half a point to a point and a half. Whenever a configuration below changes the optimizer, retune the learning rate.

    1. Train both recipes on `FashionMNIST10k` five times each and estimate the standard deviation of the validation accuracy across runs. How large must the difference between two configurations be before one run of each resolves it?
    2. Starting from the compact modern recipe, replace one ingredient at a time by its recipe A counterpart: remove Mixup, remove label smoothing, replace the warmup and cosine decay by recipe A's step decay, replace AdamW by recipe A's SGD with momentum, and shorten the budget from 45 to 30 epochs. Which of the five changes in validation accuracy exceed the threshold of (a), and do the five add up to the total gap?

3. Train the compact modern recipe with both the optimizer and the schedule of recipe A. Together with the compact modern recipe and the optimizer and schedule replacements of (b), this run fills a 2×2 table of optimizer against schedule. Do the effects of the two ingredients interact?

5. code **Weight decay and batch normalization.** ▯ The training refinements collected by He et al. (2019) apply weight decay (Section 2.7) only to the weights of convolutional and linear layers and exempt every bias and every scale γ and shift β of batch normalization.

   1. When batch statistics are used, the output of batch normalization applied to a convolution is unchanged if the convolution's weights $\mathbf{W}$ are multiplied by any $c > 0$. What does weight decay on $\mathbf{W}$ then do to the loss, to $\|\mathbf{W}\|$ over the course of training, and to the effective step size of each gradient update?
   2. Implement both variants, decaying every parameter and decaying only the convolutional and linear weights, on `FashionMNIST10k` under the compact modern recipe. Compare validation accuracy and plot the norms of the exempted parameters over training.

   Adapted from Simon Prince, Understanding Deep Learning, Problem 11.7.

6. code **Stochastic depth in the recipe.** ▯ The compact modern recipe omits stochastic depth. Add it back.

   1. Extend `StochasticResidual` to the downsampling blocks and build the ResNet-18 of Section 7.4 from it, with the drop probability growing linearly from 0 in the first block to p_{max} in the last. What is the expected number of residual branches kept ($b = 1$) for one training example, as a function of p_{max}?
   2. Sweep p_{max} over 0.1, 0.2, and 0.5 on `FashionMNIST10k` under the compact modern recipe, which is itself the case $p_{\text{max}} = 0$. Does any value raise validation accuracy by more than the run-to-run spread of Table 7.5? How does the best value compare with the range of 0.1 to 0.5 used for much deeper networks?
   3. The per-example implementation of `StochasticResidual` computes every branch before masking it. Implement the original per-minibatch variant of Huang et al. (2016), which draws one b in Equation 7.10 per block and step and skips the branch entirely when $b = 0$. Use (a) to predict the fraction of residual branches it skips at $p_{\text{max}} = 0.5$. Then measure the fraction of training time it saves and compare the two.

7.7 ConvNeXt: Modernizing a ResNet Step by Step

Every network of this chapter so far is a convolutional network, and until 2020 the best image classifiers were convolutional networks too. In 2021 a network without convolutions overtook them. The Swin Transformer (Liu et al. 2021), built around a layer called *attention*, reached 81.3% top-1 accuracy on ImageNet, against 76.1% for a ResNet-50 of similar cost. Did attention cause the gap? If it did, convolutional networks were obsolete.

A sketch of the newcomer is all we need here; Chapter 11 treats it properly. A *vision transformer* (Dosovitskiy et al. 2021) cuts the image into a grid of patches, turns each patch into a vector, and passes these vectors through a stack of identical blocks. Each block has two parts. An attention layer lets every patch take a weighted average of the features of other patches, with weights computed from the features themselves; a small MLP then transforms

each patch's vector on its own. Swin computes attention within local 7×7 windows of patches and lowers the resolution between stages as a convnet does.

The previous section suggests caution. Swin differs from a 2015 ResNet in many ways besides attention: it was trained with a newer recipe, its stem slices the image into patches, and it uses another activation, another normalization, and other stage depths. Which of these differences carried the improvement? Z. Liu et al. (2022) answered by experiment. Starting from a ResNet-50, they applied one Swin-inspired change at a time, measured ImageNet accuracy after each, and never added attention. The result, *ConvNeXt*, reaches 82.0%. Its ingredients come from earlier sections: the training recipe of Section 7.6, the depthwise convolutions and inverted bottlenecks of Section 7.5, and the layer normalization of Section 7.3. Every row of their table is a full ImageNet training run, so we cannot repeat the experiment. We read the table instead and build the network it ends in. Then we run a small roadmap of our own on Fashion-MNIST: starting from a ResNet-18 with the same number of parameters, we apply the paper's changes in three steps and train after each.

7.7.1 The Modernization Roadmap

Table 7.6 lists the changes in the order the paper applied them. Each row keeps every change above it, so the accuracies are cumulative, and the computational cost stays near that of Swin-T, about 4.5 billion multiply-adds per image (the paper's 4.5 GFLOPs).

Table 7.6: The ConvNeXt modernization roadmap (Z. Liu et al. 2022). ImageNet top-1 accuracy of a ResNet-50 as design changes accumulate; each row includes all rows above it, at roughly constant cost. Swin-T, a transformer of similar cost, reaches 81.3%.

Change	What it means	Top-1
(starting point)	ResNet-50 with its 2015 recipe	76.1%
Modern training recipe	300 epochs, AdamW, Mixup, label smoothing, stochastic depth (Section 7.6)	78.8%
Stage ratio	blocks per stage $(3, 4, 6, 3) \rightarrow (3, 3, 9, 3)$	79.4%
Patchify stem	7×7 stride-2 convolution plus pooling $\rightarrow$ one 4×4 stride-4 convolution	79.5%
Depthwise convolutions	the 3×3 convolution becomes depthwise; width grows from 64 to 96 channels	80.5%
Inverted bottleneck	expand $4\times$ inside the block instead of compressing	80.6%
Large kernels	depthwise convolution moved to the front and enlarged to 7×7	80.6%
ReLU $\rightarrow$ GELU	a smooth activation, as in transformers	80.6%
Fewer activations	one activation per block instead of three	81.3%

Change	What it means	Top-1
Fewer normalizations	one normalization per block instead of three	81.4%
BN $\rightarrow$ LN	layer normalization over channels at each position	81.5%
Separate downsampling	an LN plus 2×2 stride-2 convolution between stages	82.0%

The first change gives the largest single improvement, and it does not touch the architecture at all. This is the recipe effect of Section 7.6, here worth 2.7 points on an unchanged ResNet-50. Of the 5.9 points that separate the 2015 ResNet-50 from ConvNeXt, 2.7 thus come from training and 3.2 from the architecture, at least in this order of application. Papers that compared a well-tuned transformer against a ResNet trained the 2015 way were therefore comparing recipes as much as architectures.

7.7.1.1 Macro Design

The next two rows change the shape of the network as a whole. Swin-T distributes its blocks across four stages in the ratio $1{:}1{:}3{:}1$, spending most of its depth where the feature maps are small and each block is cheap. Adopting the same ratio turns ResNet-50's $(3, 4, 6, 3)$ blocks into $(3, 3, 9, 3)$ and adds 0.6 points. The stem changes more. ResNet begins with a 7×7 stride-2 convolution followed by max-pooling, a fourfold reduction computed with overlapping windows. A vision transformer instead slices the image into non-overlapping patches and multiplies the pixels of each patch by one shared matrix. That is nothing more exotic than a convolution whose kernel size equals its stride (Section 6.3), here a 4×4 convolution with stride 4. This *patchify* stem replaces the ResNet stem with no loss (79.5%) and produces one feature vector per 4×4 patch, the same object a transformer computes.

7.7.1.2 Depthwise Convolutions and the Inverted Bottleneck

A transformer block separates two kinds of mixing. Attention combines information across positions, and the MLP that follows combines the channels of each position on its own. A convolutional network can separate them the same way, with the two layers of Section 7.5.1.1: a *depthwise* convolution mixes positions but not channels, and a 1×1 convolution mixes channels at each position but not positions. ConvNeXt makes the 3×3 convolution of the ResNet bottleneck depthwise. That cuts its cost so much that the network can afford more width: 96 channels in the first stage instead of 64, matching Swin-T, which brings 80.5%.

Why does the ResNet bottleneck compress its input before the spatial convolution and expand it afterwards? Because the cost of a dense convolution grows with the square of the width. The cost of a depthwise convolution grows only in proportion to the width, so the reason to compress is gone, and the block can keep a thin stream between blocks and expand it inside, as MobileNetV2 does (Section 7.5). The transformer's MLP has the same shape, a 1×1 expansion by $4\times$, an activation, and a projection back. *Inverting* the bottleneck this way adds a little (80.6%).

With the dense convolution gone, the kernel size becomes cheap to change. The depthwise convolution moves to the front of the block, so that it acts on the thin stream of c channels

instead of the $4c$ expanded ones. This alone costs 0.7 points (to 79.9%), the only step backwards on the roadmap, and enlarging the kernel from 3×3 to 7×7 recovers them. A 7×7 kernel widens the receptive field as much as three stacked 3×3 layers (Section 6.3). VGG chose the stack (Section 7.2.1) because dense kernels cost $k^2 c_{\text{i}} c_{\text{o}}$ multiply-adds per position. A depthwise kernel costs $k^2 c$, so going from 3×3 to 7×7 multiplies a small term by $49/9 \approx 5$. The authors report no further gain from kernels beyond 7×7, which is also the window size of Swin's attention.

7.7.1.3 Micro Design

The remaining rows change individual layers. GELU (Hendrycks and Gimpel 2016), the smooth variant of the ReLU that transformers use, changes nothing by itself (80.6%). What matters is *how many* activations and normalizations the block contains. A ResNet bottleneck follows every convolution with a normalization and an activation: three of each. A transformer block has one activation, inside its MLP, and one normalization in front of each of its two parts. Keeping only the GELU between the two 1×1 convolutions adds 0.7 points, more than any other change in this group. Why should a network improve when it loses two of its three nonlinearities? We know of no convincing argument, and the gain is best taken as an empirical finding. Keeping only one normalization adds 0.1, and making that one a layer normalization over the channels at each spatial position (Section 7.3.4.1), instead of a batch normalization, adds another tenth (81.5%). Section 7.3 introduced normalization as an aid to optimization; here fewer such layers, one per block, train better than three. Figure 7.19 shows the block before and after; its layer scale γ is defined with the implementation below.

Finally, ResNet downsamples inside the first block of each stage by giving its 3×3 convolution a stride of 2. Swin uses a separate patch-merging layer between stages. Adopting the same separation completes the roadmap at 82.0%: a layer normalization followed by a 2×2 stride-2 convolution between stages, plus a normalization after the stem and one before the classifier, without which training was unstable. Figure 7.20 shows the assembled ConvNeXt-T, with four stages of $(3, 3, 9, 3)$ blocks at widths $(96, 192, 384, 768)$, in the stem-body-head layout this chapter has used since Section 7.2.3.

At equal cost, ConvNeXt-T matches Swin-T without a single attention layer. The gap of 2021 came from the training recipe and from design choices other than attention. The experiment does not show that convolutions are better: under comparable tuning the two families reach similar accuracy.

7.7.2 Implementation

The roadmap removed more than it added, so the block is short.

7.7.2.1 The ConvNeXt Block

The block applies, in order, a 7×7 depthwise convolution, a layer normalization, a 1×1 convolution that expands the channels $4\times$, a GELU, and a 1×1 convolution that projects back, and it adds the result to its input. Two details of the paper's training setup come with it. *Layer scale* (Touvron, Cord, Sablayrolles, et al. 2021) multiplies the branch by a learnable vector γ, one entry per channel, initialized to 10^{-6}. (It shares its letter with the scale of batch normalization but is a parameter of its own.) Every block then starts close to the identity, and the network begins training as a shallow function. *Stochastic depth* drops the whole branch of a block at random during training; we reuse `d2l.drop_path` of Section 7.6.

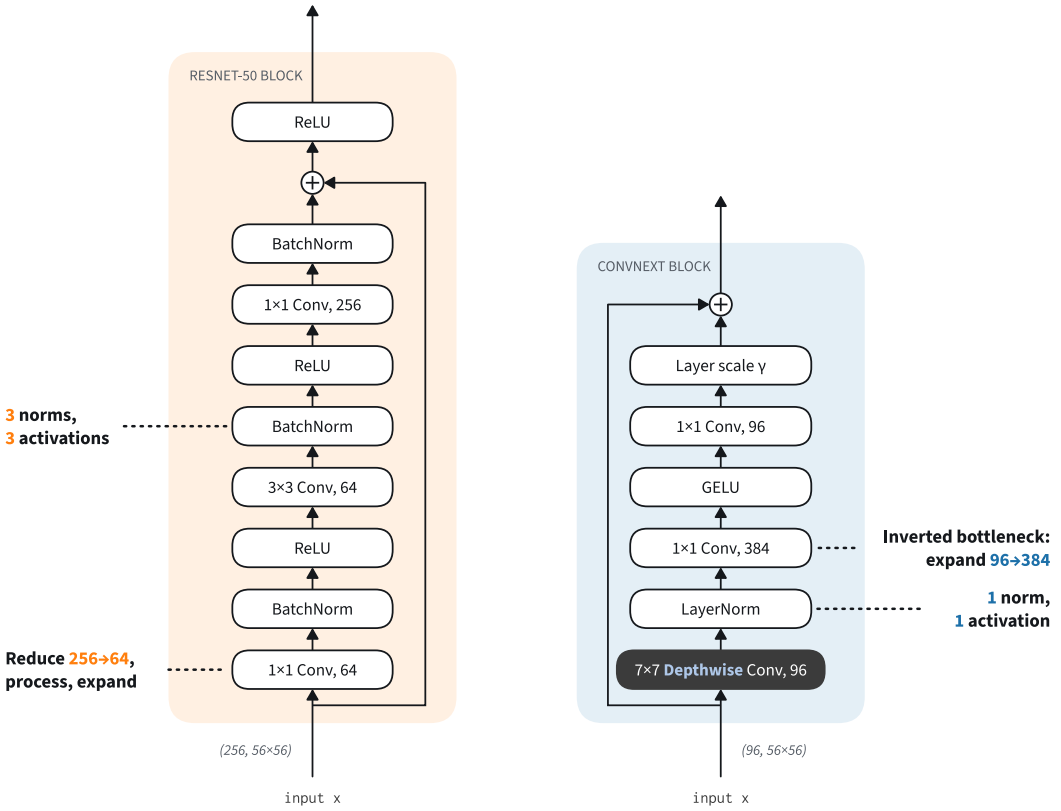


Figure 7.19: The ResNet-50 bottleneck block and the ConvNeXt block that the roadmap turns it into. Three normalizations and three activations become one of each; the compress-process-expand bottleneck becomes an inverted bottleneck led by a 7×7 depthwise convolution, and a learnable per-channel layer scale γ multiplies the branch before the addition.

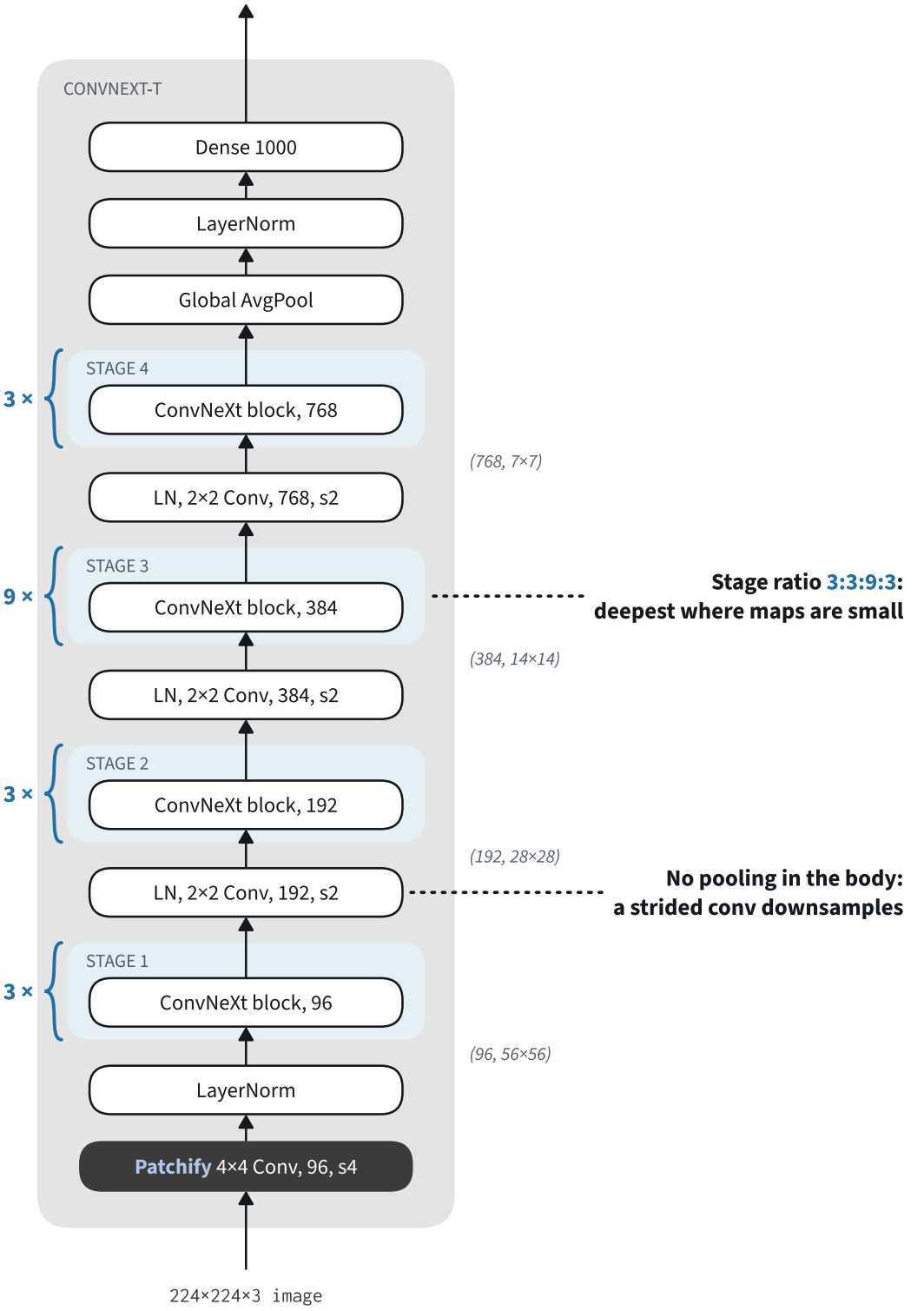


Figure 7.20: ConvNeXt-T. A patchify stem replaces convolution-plus-pooling; stages of 3, 3, 9, and 3 blocks are joined by separate LN plus 2×2 stride-2 downsampling layers; there is no pooling anywhere in the body.

The layer normalization acts on the channel vector at each spatial position. In a channels-last layout (examples, height, width, channels) this is the default layer normalization over the last axis, and a 1×1 convolution is a linear layer applied to that axis. Following the paper, every convolution and linear layer starts with weights drawn from a normal distribution with standard deviation 0.02 and zero biases.

Images are stored channels-first, so `LayerNorm2d` permutes to channels-last for the normalization and back, and the block does the same around its two linear layers.

```
class LayerNorm2d(nn.LayerNorm):
    """LayerNorm over the channel axis of an NCHW tensor."""
    def __init__(self, num_channels, eps=1e-6):
        super().__init__(num_channels, eps=eps)

    def forward(self, X):
        return super().forward(X.permute(0, 2, 3, 1)).permute(0, 3, 1, 2)

class ConvNeXtBlock(nn.Module):
    """Depthwise 7x7, norm, 1x1 expand, GELU, 1x1 project, scaled residual."""
    def __init__(self, dim, drop_prob=0.0, layer_scale=1e-6,
                 norm=LayerNorm2d):
        super().__init__()
        self.dwconv = nn.Conv2d(dim, dim, kernel_size=7, padding=3,
                                groups=dim)
        self.norm = norm(dim)
        self.pwconv1 = nn.Linear(dim, 4 * dim)  # 1x1 conv in channels-last
        self.pwconv2 = nn.Linear(4 * dim, dim)
        self.gamma = nn.Parameter(layer_scale * torch.ones(dim))
        self.drop_prob = drop_prob

    def forward(self, X):
        Y = self.norm(self.dwconv(X)).permute(0, 2, 3, 1)  # to channels-last
        Y = self.pwconv2(F.gelu(self.pwconv1(Y)))
        Y = (self.gamma * Y).permute(0, 3, 1, 2)  # back to channels-first
        return X + d2l.drop_path(Y, self.drop_prob, self.training)
```

7.7.2.2 The Full Network

The network is once more the loop over a tuple of (depth, channels) pairs of Section 7.2.1. The stem is the patchify convolution plus a layer normalization, and the downsampling layer sits between the stages; there is no pooling in the body. Following the paper, the stochastic-depth probability grows linearly from zero at the first block to its maximum at the last. For Fashion-MNIST we use the widths $(40, 80, 160, 320)$ and depths $(2, 2, 6, 2)$ of ConvNeXt-atto, the smallest variant in common use; the third stage is the deepest, as in ConvNeXt-T. The normalization layer is an argument, `norm`, and the stem a method, `stem`, so that the experiment at the end of this section can exchange them.

```
class ConvNeXt(d2l.Classifier):
    def __init__(self, arch=((2, 40), (2, 80), (6, 160), (2, 320)),
                 lr=2e-3, num_classes=10, drop_path_max=0.1,
                 norm=LayerNorm2d):
```

```
        super().__init__()
        self.save_hyperparameters()
        depths = [d for d, c in arch]
        rates = torch.linspace(0, drop_path_max, sum(depths)).tolist()
        layers = self.stem(arch[0][1], norm)
        c_prev, b = arch[0][1], 0
        for i, (depth, c) in enumerate(arch):
            if i > 0:  # Separate downsampling layer between stages
                layers += [norm(c_prev),
                           nn.Conv2d(c_prev, c, kernel_size=2, stride=2)]
            for _ in range(depth):
                layers.append(ConvNeXtBlock(c, rates[b], norm=norm))
                b += 1
            c_prev = c
        layers += [nn.AdaptiveAvgPool2d((1, 1)), norm(c_prev), nn.Flatten(),
                   nn.Linear(c_prev, num_classes)]
        self.net = nn.Sequential(*layers)
        self.net.apply(self.weight_init)

    def stem(self, c, norm):  # Patchify: a 4x4 convolution with stride 4
        return [nn.Conv2d(1, c, kernel_size=4, stride=4), norm(c)]

    def weight_init(self, module):
        if type(module) in (nn.Conv2d, nn.Linear):
            nn.init.normal_(module.weight, std=0.02)
            nn.init.zeros_(module.bias)
```

We train on Fashion-MNIST upsampled to 64×64. Such an input leaves the stem as a 16×16 map, and the three downsampling layers reduce it to 8×8, 4×4, and finally 2×2 before the head pools it away. We check the output shape and count the parameters.

```
model = ConvNeXt()
assert model(torch.randn(1, 1, 64, 64)).shape == (1, 10)
d2l.num_params(model)
```

```
3376450
```

The network has 3,376,450 parameters, about a third of the 11.2 million of the ResNet-18 that Section 7.4 trained on Fashion-MNIST. Because a single wrongly sized layer changes it, the count is a sensitive check of any reimplementation, ours included.

7.7.3 A Small Roadmap on Fashion-MNIST

Every choice on the roadmap was made on 1.3 million ImageNet images at 224×224. Which of them also help on small grayscale images? Nothing guarantees that any do. We test them the way the paper found them: start from a control, apply the changes cumulatively, and train after each step. The control has as many parameters as our ConvNeXt: a ResNet-18 whose base width is reduced from 64 to 35 channels. Every network uses the compact modern recipe B of Section 7.6 with its settings: AdamW, that is, Adam with decoupled weight decay (Chapter 9), warmup and cosine decay, label smoothing, and Mixup. Only the length differs: 20 epochs on the full training set instead of 45 epochs on 10,000 images, with a warmup over the first 10% of the steps instead of three epochs. Like recipe B, we leave out

stochastic depth, by setting ConvNeXt's rate to zero, so that the networks differ in architecture alone.

The class `ModernRecipe` below packages that recipe as a mixin, so that one copy serves every network: any classifier that stores its learning rate `lr` and its total number of optimization steps `num_steps` can inherit the optimizer, its schedule, and the training step. The validation loss stays the plain cross-entropy of `Classifier`, so it means the same in every section.

```
class ModernRecipe:
    """AdamW, linear warmup then cosine decay, label smoothing, Mixup."""
    def configure_optimizers(self):
        optim = torch.optim.AdamW(self.parameters(), lr=self.lr,
                                  weight_decay=0.05)
        warmup = self.num_steps // 10
        schedule = torch.optim.lr_scheduler.LambdaLR(
            optim, lambda t: d2l.cosine_warmup(t, self.num_steps, warmup))
        return optim, schedule  # The Trainer steps it after every minibatch

    def training_step(self, batch):
        X, y_a, y_b, lam = d2l.mixup(*batch, alpha=0.2)
        Y_hat = self(X)
        loss = lambda y: F.cross_entropy(Y_hat, y, label_smoothing=0.1)
        return lam * loss(y_a) + (1 - lam) * loss(y_b)

class ModernConvNeXt(ModernRecipe, ConvNeXt):
    def __init__(self, num_steps, lr=2e-3, norm=LayerNorm2d):
        super().__init__(lr=lr, drop_path_max=0.0, norm=norm)
        self.num_steps = num_steps
```

The control stacks the `Residual` blocks of Section 7.4 in the layout of ResNet-18: a 7×7 stride-2 stem with max-pooling, four stages of two blocks each with 35, 70, 140, and 280 channels, and a global-average-pooling head.

```
class CompactResNet18(ModernRecipe, d2l.Classifier):
    """ResNet-18 with base width 35 instead of 64, trained by the recipe."""
    def __init__(self, num_steps, lr=2e-3, num_classes=10, base=35):
        super().__init__()
        self.save_hyperparameters()
        layers = [nn.LazyConv2d(base, kernel_size=7, stride=2, padding=3,
                                bias=False),
                  nn.LazyBatchNorm2d(), nn.ReLU(),
                  nn.MaxPool2d(kernel_size=3, stride=2, padding=1)]
        for i, c in enumerate((base, 2 * base, 4 * base, 8 * base)):
            for j in range(2):
                down = i > 0 and j == 0
                layers.append(d2l.Residual(c, use_1x1conv=down,
                                           strides=2 if down else 1))
        layers += [nn.AdaptiveAvgPool2d((1, 1)), nn.Flatten(),
                   nn.LazyLinear(num_classes)]
        self.net = nn.Sequential(*layers)

    def weight_init(self, module):
```

```
        # The Trainer calls this after the first minibatch fixed the shapes
        if type(module) in (nn.Conv2d, nn.Linear):
            nn.init.kaiming_normal_(module.weight, nonlinearity='relu')
            if module.bias is not None:
                nn.init.zeros_(module.bias)
```

Our roadmap groups the paper's changes by the part of the network they act on. The first step replaces the blocks and stages of the ResNet by those of ConvNeXt (the 7×7 depthwise convolution in an inverted bottleneck, one GELU and one normalization per block, layer scale, the $(2, 2, 6, 2)$ stages, and the separate downsampling layers) but keeps the ResNet stem and batch normalization; it covers eight rows of Table 7.6. The second step replaces batch normalization by layer normalization. The third replaces the ResNet stem by the patchify stem, which completes our ConvNeXt. The paper changed the stem before the blocks; the final network does not depend on the order, although the gain credited to each step can. In code, the steps are the argument `norm` and a subclass, `ResNetStemConvNeXt`, that overrides the method `stem`.

```
class ResNetStemConvNeXt(ModernConvNeXt):
    """ConvNeXt with the stem of ResNet-18 instead of the patchify stem."""
    def stem(self, c, norm):  # 7x7 convolution with stride 2, max-pooling
        return [nn.Conv2d(1, c, kernel_size=7, stride=2, padding=3), norm(c),
                nn.ReLU(), nn.MaxPool2d(kernel_size=3, stride=2, padding=1)]

roadmap = {'compact ResNet-18': CompactResNet18,
           'ConvNeXt blocks and stages, BN': (
               lambda num_steps: ResNetStemConvNeXt(num_steps,
                                                    norm=nn.BatchNorm2d)),
           '+ BN -> LN': ResNetStemConvNeXt,
           '+ patchify stem = ConvNeXt': ModernConvNeXt}
```

We train the four networks in turn and record the parameters and the validation accuracy after the last epoch of each.

```
data = d2l.FashionMNIST(batch_size=128, resize=(64, 64))
num_steps = 20 * len(data.train_dataloader())
results = {}
for name, make_model in roadmap.items():
    model = make_model(num_steps)
    trainer = d2l.Trainer(max_epochs=20, num_gpus=1)
    trainer.board.display = False  # Four runs; we tabulate the results
    trainer.fit(model, data)
    results[name] = (d2l.num_params(model), trainer.history['val_acc'][-1])
```

```
for name, (params, accuracy) in results.items():
    print(f'{name:34s} {params:>9,} parameters, '
          f'validation accuracy {accuracy:.3f}')
```

```
compact ResNet-18                  3,346,675 parameters, validation accuracy 0.939
ConvNeXt blocks and stages, BN     3,377,770 parameters, validation accuracy 0.937
+ BN -> LN                         3,377,770 parameters, validation accuracy 0.937
```

```
+ patchify stem = ConvNeXt        3,376,450 parameters, validation accuracy 0.916
```

Table 7.7: Our roadmap on Fashion-MNIST beside the paper's. The last column is the validation accuracy after 20 epochs of the modern recipe at 64×64, rounded to the stability of repeated runs; the paper's column adds up the ImageNet top-1 gains of the rows of Table 7.6 that each step covers.

Step	Rows of the paper	Gain on ImageNet	Parameters	Validation accuracy
Compact ResNet-18	(ResNet-50 with the modern recipe)		3.35M	≈ 94%
ConvNeXt blocks and stages, BN	stage ratio to fewer normalizations, separate downsampling	+3.0	3.38M	≈ 94%
BN → LN	BN → LN	+0.1	3.38M	≈ 94%
Patchify stem (our ConvNeXt)	patchify stem	+0.1	3.38M	≈ 91.5%

The changes that make up the ConvNeXt block transfer. With the ResNet stem, the ConvNeXt is as accurate as the ResNet-18, with batch or with layer normalization: the first three rows agree within the spread of repeated runs, a few tenths of a point. The eight rows that were worth three points on ImageNet gain nothing here, and they lose nothing either. All of the loss, about two points, comes from the one step that was neutral on ImageNet, the patchify stem. The stem it replaces convolves at twice the final resolution and then max-pools over neighboring positions; the patchify stem reaches the 16×16 grid in a single linear step, without pooling. Which of these differences matters we have not tested.

The control is fair as far as arithmetic goes: counting the multiply-adds of every convolution and linear layer for one 64×64 image gives 43.2 million for the ResNet, 46.2 million for the ConvNeXt with the ResNet stem, and 44.3 million for our ConvNeXt. We did not match the running time, which multiply-adds do not determine (Section 7.5), and 20 epochs are short for a design tuned for 300. Within these limits the experiment locates the two points that our ConvNeXt loses. Design choices selected on ImageNet do not transfer automatically, but here only one of them fails to.

7.7.4 Limits and Open Problems

ConvNeXt answers one question, which design choices closed the gap of 2021, and leaves two open. The first is pretraining. Large vision models are now often pretrained without labels, for instance as *masked autoencoders* (He et al. 2022) that hide most patches of an image and learn to reconstruct them. That is natural for a network that treats the image as a set of patches and awkward for a convolution that slides across the holes. ConvNeXt V2 (Woo et al. 2023) makes it work; problem 5 implements the layer it adds. The second is how far a convolution needs to see. A 7×7 kernel sees a small window, whereas global attention lets every position read from every other. Whether much larger kernels (Ding et al. 2022), convolutions that learn where to sample, or attention is the better way to see the whole image at a given cost is still open.

7.7.5 Summary

ConvNeXt is a controlled experiment wearing an architecture's name. Starting from a ResNet-50, it applied the design choices of the Swin Transformer one at a time and kept those that helped: a modern training recipe, a $1{:}1{:}3{:}1$ stage ratio, a patchify stem, depthwise convolutions in an inverted bottleneck led by a 7×7 kernel, one GELU and one layer normalization per block, and separate downsampling layers. The result matches Swin-T at equal cost without attention, and almost half of its gain over the 2015 ResNet-50 comes from the recipe alone. Its block separates mixing across positions from mixing across channels, as a transformer block separates attention from its MLP. Choices selected at ImageNet scale need not transfer, and a small roadmap shows which do not: on upsampled Fashion-MNIST the ConvNeXt block, its stages, and layer normalization match a ResNet-18 with the same number of parameters, and the patchify stem alone costs about two points.

7.7.6 Exercises

1. **Roadmap ablations.** The roadmap in Table 7.6 applies its changes in one fixed order, and each row's gain is measured with every earlier change in place.
    1. Just before the "Large kernels" row, each block expands its residual width c to $4c$ with a 1×1 convolution, applies a 3×3 depthwise convolution to the $4c$ channels, and projects back to c with a second 1×1 convolution. The row first moves the depthwise convolution in front of the expansion, where it acts on c channels, and then enlarges its kernel to 7×7. Count the multiply-adds per spatial position of the depthwise convolution before the move, after the move, and after the enlargement. Compare the net change over the row with the change that enlarging the kernel to 7×7 in place, on the $4c$ channels, would have caused.
    2. Choose two rows whose gains you expect to depend on their order, and design the additional runs that would measure the interaction.
2. **Where the parameters live.**
    1. For the `ConvNeXt` of this section, compute the fraction of parameters in the depthwise convolutions, in the 1×1 expansions and projections, in the normalizations and layer-scale vectors inside the blocks, in the downsampling layers, and in the stem and head. Check that the total is 3,376,450.
    2. Express the parameter count of a `ConvNeXtBlock` and of a `Residual` block of Section 7.4 without the 1×1 shortcut as functions of the width c. Which design decision makes the ConvNeXt block cheaper, and by what factor for large c?
3. **Normalization axes.** The activations entering a normalization layer form a minibatch of n images, each with c channels on an $h \times w$ map. Consider three layers: batch normalization, the per-position layer normalization of `ConvNeXtBlock`, and a layer normalization over all chw values of an image with one learnable scale and one learnable shift per value.
    1. For each layer, state which values share one mean and variance, and count its learnable parameters.
    2. A network trained on 64×64 images is evaluated on 224×224 images. Which of the three layers can be applied to the larger images with their trained parameters unchanged? Explain.
4. code **Layer scale.** Under the recipe of the roadmap in Table 7.7, train `ConvNeXt` with γ

initialized to 10^{-6} (the default), with γ initialized to 1, and with layer scale removed, and compare the training curves. Layer scale at 10^{-6} and stochastic depth (Equation 7.10 of Section 7.6) both make the network effectively shallower than its block count. For each mechanism, explain how it does so, state when during training it acts, and state whether it changes the network used at evaluation.

5. code **Global Response Normalization.** ConvNeXt V2 (Woo et al. 2023) inserts GRN after the GELU of the block and removes layer scale. For a feature map $\mathbf{X}$ with C channels, let g_c be the ℓ_2 norm of channel c over all spatial positions and $\bar{g}$ the mean of $g_1, \dots, g_C$. GRN rescales each channel by its normalized response and adds a learnable per-channel affine term and a residual,

$$\mathrm{GRN}(\mathbf{X})_c = \gamma_c \frac{g_c}{\bar{g} + \epsilon} \mathbf{X}_c + \beta_c + \mathbf{X}_c, \tag{7.12}$$

with a small ϵ against division by zero (10^{-6} in the paper's code), and with γ_c and β_c initialized to zero so that the layer starts as the identity.

   1. Implement GRN in about five lines and insert it into `ConvNeXtBlock` after the GELU, dropping layer scale.
   2. Retrain under the recipe of the roadmap in Table 7.7 and compare the validation accuracy.

6. code **Depthwise kernel size.** ☒ Under the recipe of the roadmap in Table 7.7, train `ConvNeXt` with 3×3 and with 11×11 depthwise convolutions that leave the feature-map size unchanged, and compare accuracy, parameter count, and time per epoch against the 7×7 baseline. Does accuracy saturate at 7×7 here as Z. Liu et al. (2022) report on ImageNet? How does the change in time per epoch compare with the change in the depthwise term's arithmetic, and why?

7. code **Channel redundancy.** ☒ Take the `ConvNeXt` trained with GRN in problem 5 and the one trained without it under the same recipe. Measure the channel redundancy that GRN was designed to prevent, at the input of the second 1×1 convolution of every block. For each validation image, flatten each channel's map into a vector over the spatial positions, and average the cosine similarity over all pairs of channels. Average over the images and compare the networks with and without GRN block by block. In which stages, if any, does GRN make the channels less similar?

8. code **Recipe downgrade.** ☒ The first and last rows of Table 7.7 train two architectures under the modern recipe. Complete the ablation square by training both models under recipe A of Section 7.6: SGD with momentum, step decay, and neither Mixup nor label smoothing, with the learning rate retuned. How much of each network's accuracy survives the downgrade, and does the recipe help both architectures equally? Compare the size of the effect with the 2.7 points that the "Modern training recipe" row of Table 7.6 attributes to the recipe on ImageNet.

7.8 Convolutional Network Design Spaces

Most networks in this chapter were designed by hand. Each design fixes a few numbers for every stage: how many blocks it repeats, how many channels they produce, and what the block looks like. Where do these numbers come from, and are there rules for choosing them? The networks we built give little guidance. ResNet-18 repeats its blocks $(2, 2, 2, 2)$

times across four stages, ResNet-50 $(3, 4, 6, 3)$ times, and ConvNeXt-T $(3, 3, 9, 3)$ times. ResNet's depths come with no stated rule, and ConvNeXt-T's were copied from Swin and checked once. The choices interact: at a fixed budget of computation, a deeper stage must be narrower, and a block with grouped convolutions can afford more channels than one without. Could we not simply try every combination? Each stage of the networks below is described by four numbers, and with four stages and only two values for each number there are already $2^{16} = 65{,}536$ variants, each needing its own training run. Worse, a single good network, however it was found, does not say which of its choices mattered, so nothing carries over to a new compute budget or a new block.

Neural architecture search, which found EfficientNet's base network (Section 7.5), automates the hand design (Zoph and Le 2016; Liu et al. 2018): it searches a set of candidate networks for the one with the best estimated accuracy. Like hand design, however, it returns one network rather than a reason, and compound scaling only enlarges that network. Radosavovic et al. (2020) take a different route and compare whole *design spaces*: populations of networks defined by a template and a few constraints. They sample networks from a space, train each briefly, and keep a constraint if it improves the error distribution of the whole population. The constraints that survive are design rules, and the networks that satisfy all of them form the RegNet family. We build the template, read the paper's evidence from ImageNet, where each comparison trains 500 networks, and then compare two small populations of our own on Fashion-MNIST that differ in the rule the paper found to matter most. Finally, we train a small RegNet.

7.8.1 The AnyNet Design Space

The networks of this chapter share one template, the stem, body, and head of Section 7.2.3. The stem processes the raw image with one or a few convolutions and usually lowers its resolution. The body does most of the work. It is a sequence of *stages*: each stage operates at one resolution and repeats one kind of *block*, and each new stage halves the resolution and widens the channels. The head, global average pooling and a linear layer, turns the last feature map into class scores. ResNet's body has four stages, which end at $1/4$, $1/8$, $1/16$, and $1/32$ of the input resolution. One reason why this layout has persisted from ResNet to ConvNeXt is that detection and segmentation networks read features from all four stages, coarse ones for large objects and fine ones for small objects (Section 32.5).

7.8.1.1 Stage Parameters of Hand-Designed Networks

Table 7.8 lists the stages of four networks from this chapter. Their widths double from stage to stage in every row. Their depths follow no visible rule.

Table 7.8: Stage depths and widths of four hand-designed networks (ImageNet versions).

Network	Blocks per stage	Output channels per stage	Block
ResNet-18 (Section 7.4)	2, 2, 2, 2	64, 128, 256, 512	two 3×3 convolutions
ResNet-50 (He et al. 2016)	3, 4, 6, 3	256, 512, 1024, 2048	1×1, 3×3, 1×1 bottleneck
ResNeXt-50 (Section 7.4.5.1)	3, 4, 6, 3	256, 512, 1024, 2048	bottleneck with 32 groups

Network	Blocks per stage	Output channels per stage	Block
ConvNeXt-T (Section 7.7)	3, 3, 9, 3	96, 192, 384, 768	7×7 depthwise, inverted bottleneck

The *AnyNet* design space of Radosavovic et al. (2020) fixes the template and leaves these numbers free (Figure 7.21). The stem is one 3×3 convolution with stride 2 and $c_0 = 32$ output channels, followed by batch normalization and a ReLU. In the paper it reads RGB images; ours reads the single channel of Fashion-MNIST. The body has four stages, and stage i repeats one block d_i times. The block is the ResNeXt block of Figure 7.11: a 1×1 convolution narrows the c_i channels of the stage to c_i/b_i, a 3×3 convolution processes them in groups of g_i channels each, and a second 1×1 convolution restores c_i channels. The symbol b_i is the paper's, for the *bottleneck ratio*; the number of bottleneck channels, which Section 7.4.5.1 called b, is c_i/b_i here. Batch normalization follows each convolution, and a residual connection adds the input. The first block of each stage halves the resolution with stride 2 and passes its shortcut through a 1×1 convolution so that the shapes match. The head is global average pooling and one fully connected layer.

Each stage has four free numbers: its depth d_i, its width c_i, the bottleneck ratio $b_i \geq 1$, and the group width g_i, the number of channels per group. A warning about g_i is in order: the paper counts channels per group, whereas g in Section 7.4.5.1 counted groups, and the `groups` argument of `d2l.ResNeXtBlock` keeps the latter meaning. Four stages give 16 numbers. The paper samples depths up to 16, widths up to 1024 in multiples of 8, bottleneck ratios from $\{1, 2, 4\}$, and group widths from the powers of two up to 32, which makes about $(16 \cdot 128 \cdot 3 \cdot 6)^4 \approx 2 \times 10^{18}$ networks: far too many to enumerate, but easy to sample from. For 224×224 ImageNet images the stem and the four stages each halve the resolution, so the body ends at $224/2^5 = 7$ pixels on a side.

7.8.1.2 Implementing AnyNet

To work with a design space, we need code that builds any of its members from the stage parameters. We start with the stem, a three-layer sequential block.

```
class AnyNet(d2l.Classifier):
    def stem(self, num_channels):
        return nn.Sequential(
            nn.LazyConv2d(num_channels, kernel_size=3, stride=2, padding=1,
                          bias=False),
            nn.LazyBatchNorm2d(), nn.ReLU())
```

A stage repeats `d2l.ResNeXtBlock` from Section 7.4.5.1. Its arguments are the width c_i, the number of groups, and `bot_mul` $= 1/b_i$; the group width is the bottleneck width c_i/b_i divided by the number of groups. Only the first block of the stage has stride 2.

```
@d2l.add_to_class(AnyNet)
def stage(self, depth, num_channels, groups, bot_mul):
    blk = []
    for i in range(depth):
        if i == 0:
```

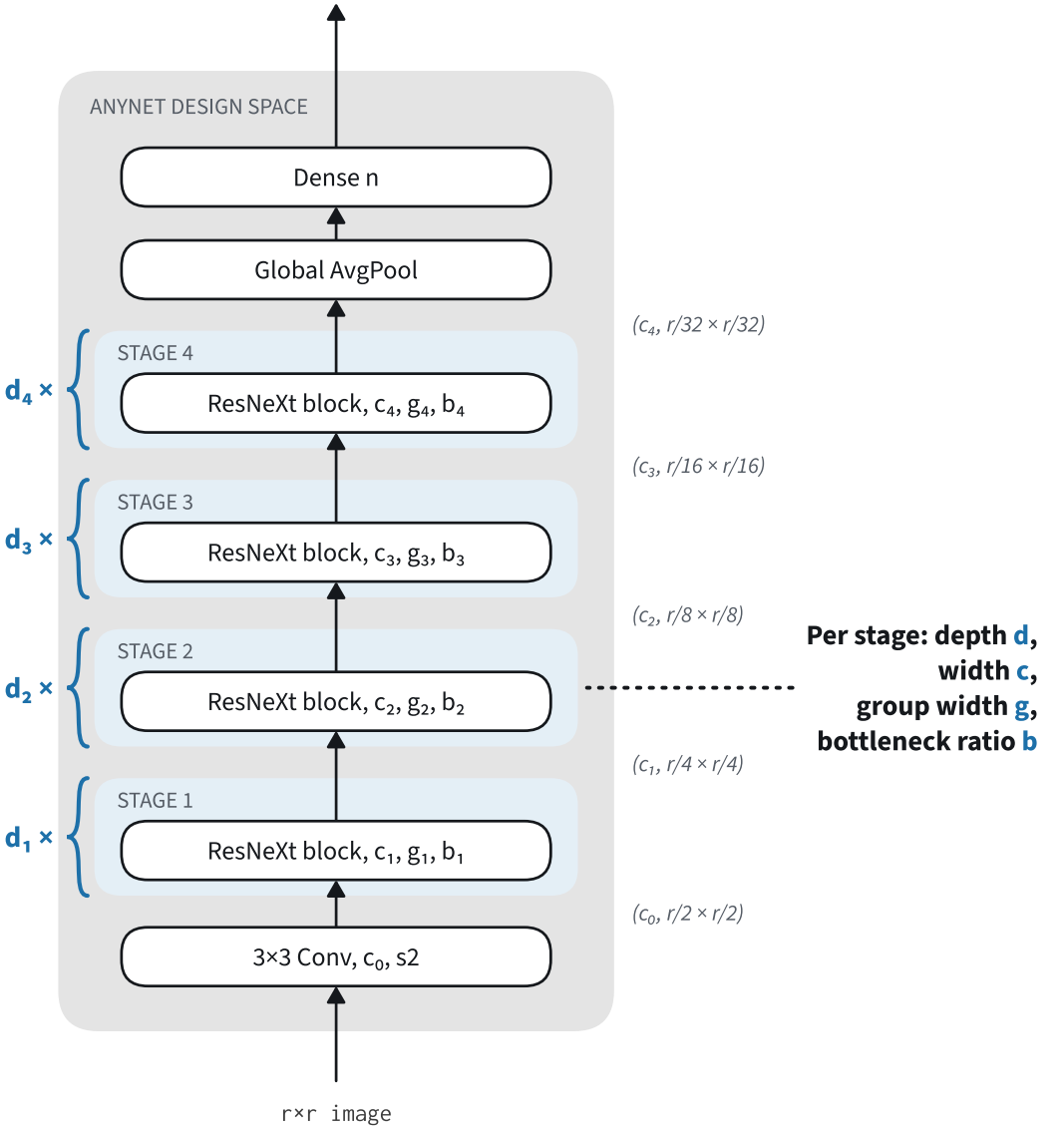


Figure 7.21: The AnyNet design space: a stem, a body of four stages, and a head. Stage i repeats a ResNeXt block d_i times; the first block of each stage halves the resolution. The annotations give the number of channels and the resolution after each part for an $r \times r$ input; the head outputs n class scores. The free choices per stage are the depth d_i, the width c_i, the group width g_i, and the bottleneck ratio b_i.

```
            blk.append(d2l.ResNeXtBlock(num_channels, groups, bot_mul,
                use_1x1conv=True, strides=2))
        else:
            blk.append(d2l.ResNeXtBlock(num_channels, groups, bot_mul))
    return nn.Sequential(*blk)
```

The network is the stem, one stage per entry of the tuple `arch`, and the head. Each entry of `arch` holds the four numbers `(depth, num_channels, groups, bot_mul)` of one stage. As in Section 7.4, every convolution and linear layer starts from He initialization (Section 4.4.2.2), and the convolutions have no bias, since batch normalization follows each of them.

```
@d2l.add_to_class(AnyNet)
def __init__(self, arch, stem_channels, lr=0.1, num_classes=10):
    super(AnyNet, self).__init__()
    self.save_hyperparameters()
    self.net = nn.Sequential(self.stem(stem_channels))
    for i, s in enumerate(arch):
        self.net.add_module(f'stage{i+1}', self.stage(*s))
    self.net.add_module('head', nn.Sequential(
        nn.AdaptiveAvgPool2d((1, 1)), nn.Flatten(),
        nn.LazyLinear(num_classes)))

@d2l.add_to_class(AnyNet)
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')
        if module.bias is not None:
            nn.init.zeros_(module.bias)
```

Every network of the space is trained with one recipe, so we attach the optimizer to `AnyNet` itself. It is the one under which Radosavovic et al. (2020) found their rules, and that of recipe A in Section 7.6: SGD with momentum 0.9, whose steps follow a running sum of recent gradients (Section 7.5; Chapter 9 treats it fully), and a weight decay that is added to the gradient (Section 2.7). The learning rate follows the schedule of this chapter (`d2l.cosine_warmup` of Section 7.1): linear warmup over the first tenth of the `num_steps` updates to the peak rate `lr`, then a cosine decay to zero. The paper uses a decay of 5×10^{-5} for 100 epochs of ImageNet. For our short runs on Fashion-MNIST we use the larger 10^{-3}, the best for the RegNetX32 of Section 7.8.3 among the handful of values we tried and as good as any for ResNet-18.

```
@d2l.add_to_class(AnyNet)
def configure_optimizers(self):
    optim = torch.optim.SGD(self.parameters(), lr=self.lr, momentum=0.9,
                            weight_decay=1e-3)  # Decay added to the gradient
    warmup = self.num_steps // 10
    schedule = torch.optim.lr_scheduler.LambdaLR(
        optim, lambda t: d2l.cosine_warmup(t, self.num_steps, warmup))
    return optim, schedule  # The Trainer steps it after every minibatch
```

7.8.2 Comparing Design Spaces

Two design spaces can be compared without finding the best network in either. Radosavovic et al. (2019) make four assumptions that let a population of briefly trained networks stand in for the space:

1. Good design principles exist, so that many networks satisfying them perform well. A distribution over networks is then a sensible object to optimize: there are many good needles in the haystack.
2. Accuracy after a few passes through the data ranks networks nearly as accuracy after full training does. This is why the paper trains each network for only 10 epochs, and we train ours below for one.
3. Results for small networks carry over to larger ones, so the search can run at a low compute budget and be checked at a high one at the end.
4. The effects of different design choices are roughly independent, so each can be studied separately.

None of the four is obvious, and the second and third are rather bold; the exercises test them at small scale.

Granting them, we sample n networks from a space, train each, and record its error. The *empirical cumulative distribution function* (CDF) of the errors $e_1, \dots, e_n$ gives, for every threshold e, the fraction of sampled networks whose error is at most e:

$$\hat{F}(e) = \frac{1}{n} \sum_{j=1}^{n} \mathbf{1}(e_j \leq e). \tag{7.13}$$

A curve that rises early belongs to a space in which many networks reach low error. If the curve of one space lies above that of another at every e, a random network from the first space is at least as likely as one from the second to beat any threshold. This compares populations, not best networks: the best network of the two spaces may be the same.

Radosavovic et al. (2020) sample 500 networks from each space at a budget of about 400 million multiply-adds (the paper's 400 MFLOPs), train each for 10 epochs on ImageNet, and compare the CDFs in Figure 7.22. Each step adds one constraint to the unconstrained AnyNetX$_A$ space. Using one bottleneck ratio b for all stages (AnyNetX$_B$) and then also one group width g (AnyNetX$_C$) leaves the curve where it was, with a mean error near 49%. This is a gain in itself: the two constraints remove six of the 16 numbers at no measurable cost, and the space becomes smaller and easier to search. The constraint that matters is on the widths. Requiring them to increase across stages, $c_{i+1} \geq c_i$, lowers the mean error of the sampled networks by six points, to about 43%, while equal or decreasing widths raise it to 56% and 63%. Requiring depths to increase as well, $d_{i+1} \geq d_i$, gains another half point; constant or decreasing depths lose ground.

How far does this evidence reach? It is specific to the paper's block, budget, data, and training protocol, but the method is not, and we can apply it at our own scale to the rule that mattered most. Our two design spaces use two stages, a stem of 32 channels, and ResNeXt blocks with $b = 1$ and a group width of 16 channels. Stage i draws its depth d_i from 1 to 8 and its width c_i from the multiples of 16 up to 256. The spaces differ in one rule only: $c_1 < c_2$ in one, $c_1 > c_2$ in the other. As in the paper, the networks are matched in compute. We count the multiply-adds of a network as for AlexNet in Section 7.1, adding up the convolutions of the stem, of every block, and of the shortcuts.

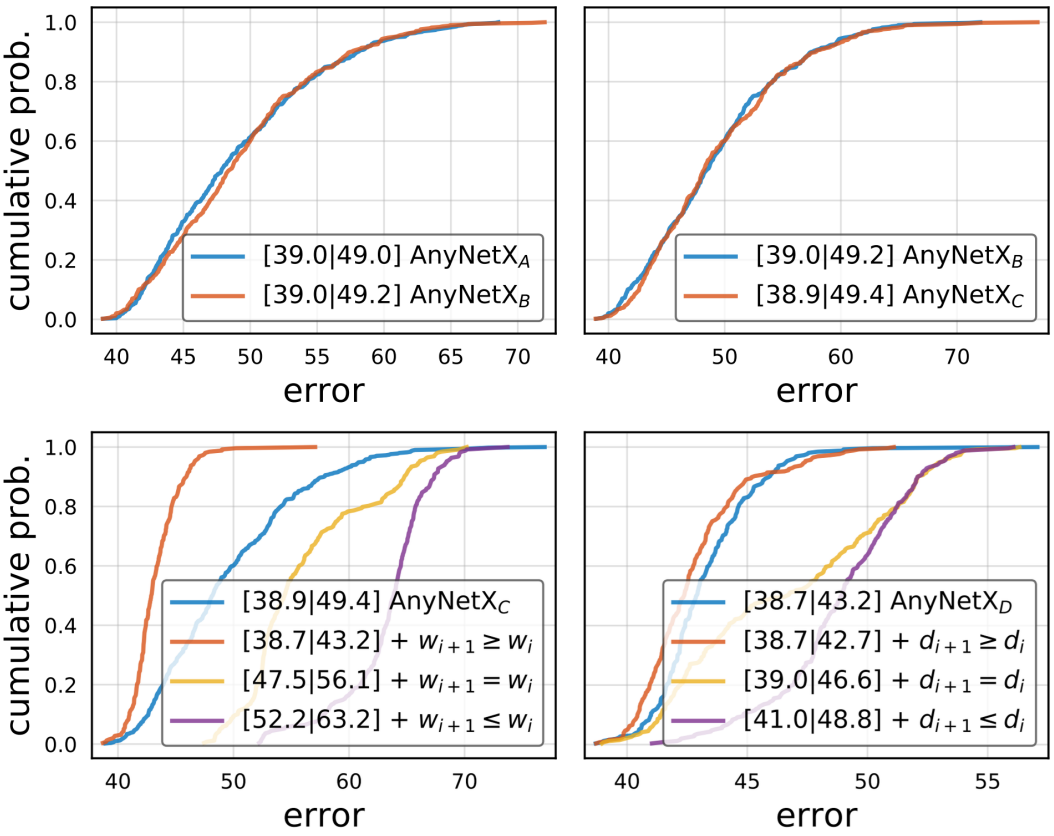


Figure 7.22: Error CDFs of design spaces, reproduced from Radosavovic et al. (2020): 500 networks sampled from each space, each trained for 10 epochs on ImageNet at about 400 million multiply-adds. Legends give the minimum and mean top-1 error in percent; w_i is the paper's symbol for the width c_i. Top: one bottleneck ratio for all stages (AnyNetX$_B$) against none (AnyNetX$_A$), then one group width as well (AnyNetX$_C$). Bottom: widths that increase, stay equal, or decrease across stages; then depths that do the same in the space with increasing widths (AnyNetX$_D$).

```
def mult_adds(arch, r, stem_channels=32, group_width=16, num_classes=10):
    """Multiply-adds of an AnyNet with k = 1 for one r x r image; arch holds
    (depth, width) for each stage."""
    r //= 2  # The stem halves the resolution
    total, c_in = r * r * 9 * stem_channels, stem_channels  # Grayscale input
    for d, c in arch:
        total += r * r * c_in * c  # First 1 x 1 convolution, before the stride
        r //= 2
        total += r * r * c_in * c  # 1 x 1 convolution of the shortcut
        total += r * r * c * c * (2 * d - 1)  # The other 1 x 1 convolutions
        total += r * r * 9 * group_width * c * d  # Grouped 3 x 3 convolutions
        c_in = c
    return total + c_in * num_classes  # Linear head
```

We sample a network by drawing two depths and two different widths, ordering the widths as the space requires, and rejecting the draw unless its multiply-adds lie within 20% of a budget. The budget is 112 million per 96×96 image, that of the RegNetX32 we train in Section 7.8.3.

```
def sample_arch(rng, increasing, r=96, budget=112e6):
    """Depths and widths of two stages at about `budget` multiply-adds."""
    while True:
        depths = rng.integers(1, 9, size=2).tolist()
        widths = sorted(rng.choice(np.arange(16, 257, 16), 2, replace=False)
                        .tolist(), reverse=not increasing)
        arch = tuple(zip(depths, widths))
        if abs(mult_adds(arch, r) / budget - 1) <= 0.2:
            return arch

rng = np.random.default_rng()
for increasing in (True, False):
    arch = sample_arch(rng, increasing)
    print(arch, f'{mult_adds(arch, 96) / 1e6:.0f} million multiply-adds')
```

```
((2, 128), (1, 240)) 97 million multiply-adds
((4, 112), (1, 16)) 100 million multiply-adds
```

At equal multiply-adds, a network with decreasing widths is wide where the maps are large, and narrow where they are small. Each sampled network trains for one epoch, a tenth of the training runs of Section 7.8.3, with the optimizer of `AnyNet`, and we record its validation error.

```
def population_errors(rng, increasing, data, n, num_epochs=1):
    """Validation errors of n networks sampled from one design space."""
    errors = []
    for _ in range(n):
        arch = sample_arch(rng, increasing)
        model = AnyNet(tuple((d, c, c // 16, 1) for d, c in arch), 32)
        model.num_steps = num_epochs * len(data.train_dataloader())
        trainer = d2l.Trainer(max_epochs=num_epochs, num_gpus=1)
        trainer.board.display = False  # We plot the errors ourselves
```

```
        trainer.fit(model, data)
        errors.append(1 - float(trainer.history['val_acc'][-1]))
    return np.array(errors)
```

We sample 12 networks from each space, 24 training runs of one epoch each.

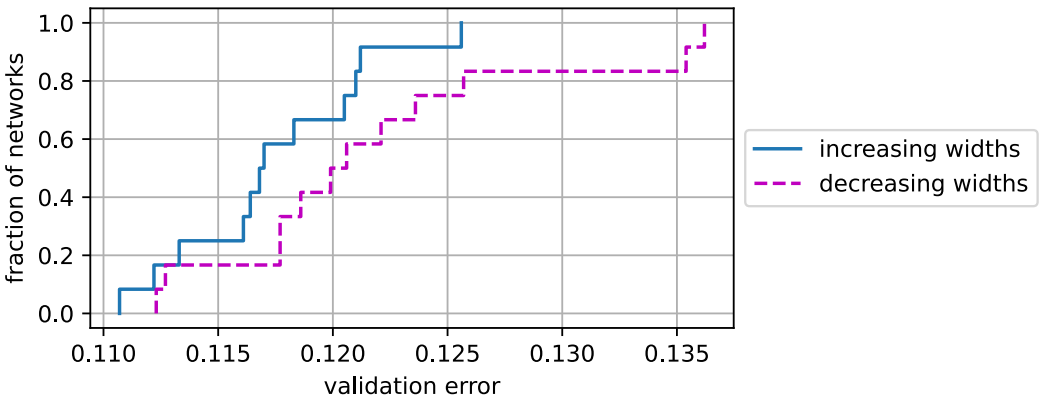
```
data = d2l.FashionMNIST(batch_size=128, resize=(96, 96))
spaces = {'increasing widths': True, 'decreasing widths': False}
errors = {name: population_errors(rng, increasing, data, 12)
          for name, increasing in spaces.items()}
```

The empirical CDF Equation 7.13 of each population is a step function that rises by $1/n$ at each sorted error.

```
def ecdf(e):
    """Corners of the step function of the empirical CDF of e."""
    e = np.sort(e)
    return np.repeat(e, 2), np.repeat(np.arange(len(e) + 1) / len(e), 2)[1:-1]

X, Y = zip(*(ecdf(e) for e in errors.values()))
d2l.plot(list(X), list(Y), xlabel='validation error',
         ylabel='fraction of networks', legend=list(errors),
         legend_loc='outside', figsize=(5, 2.5))
for name, e in errors.items():
    print(f'{name}: mean error {e.mean():.3f}, minimum {e.min():.3f}')
```

```
increasing widths: mean error 0.117, minimum 0.111
decreasing widths: mean error 0.122, minimum 0.112
```



The two curves interleave. Over repeated runs, the mean errors of the two spaces differ by less than a point, near 12%, usually in favor of increasing widths, and the best networks of both spaces reach about 11%. A rule that is worth twenty points of mean error on ImageNet is worth less than one here, and twelve samples per space cannot resolve a difference that small reliably. This is a rather sobering outcome for our first population, but not a surprising one. After one epoch on an easy dataset most networks of either space do well, which leaves the rule little to separate. Which design rules matter, and how much, depends on the data,

and the rules of Radosavovic et al. (2020) were found on ImageNet.

7.8.3 RegNet

7.8.3.1 Design Principles

The constraints that survived define the AnyNetX$_E$ space:

- one bottleneck ratio b for all stages;
- one group width g for all stages;
- widths that do not decrease across stages, $c_i \le c_{i+1}$;
- depths that do not decrease across stages, $d_i \le d_{i+1}$.

The best networks of AnyNetX$_E$ share one more regularity. If we number the blocks $j = 0, 1, \ldots$ through the whole body, the width of block j grows roughly linearly,

$$u_j = u_0 + aj,$$

with an initial width u_0 and a slope $a > 0$. Since all blocks of a stage must share one width, RegNet snaps each u_j to a nearby value on a geometric grid, whose successive values differ by a constant factor, as the widths of the hand-designed networks in Table 7.8 do. Blocks that land on the same value form a stage, which turns the line into a staircase (problem 2 works one out). The paper reports a further finding in the resulting space, and it runs against the practice of ResNet-50 and ResNeXt-50: the best networks use $b = 1$, so the block does not narrow its channels at all. The *RegNetX* family consists of the networks with quantized linear widths and $b = 1$. It is described by a handful of numbers per compute budget instead of 16.

7.8.3.2 RegNetX32 on Fashion-MNIST

We now build a small network that follows the rules and compare it with ResNet-18. Fashion-MNIST images, upsampled to 96×96 as in Section 7.4, carry only 28×28 pixels of detail, and the stem and one stage already bring them to 24×24. Four stages, as in the paper's ImageNet models, would put the last two on 6×6 and 3×3 maps. We therefore use two, which keep every block at 24×24 or 12×12 (problem 1 tries four). The stem has 32 channels; stage 1 has $d_1 = 4$ blocks of width $c_1 = 64$ and stage 2 has $d_2 = 6$ blocks of width $c_2 = 160$. We picked these numbers by hand so that widths and depths increase; nothing was sampled or searched. The bottleneck ratio is $b = 1$, and both stages share the group width $g = 16$, so their 3×3 convolutions have $64/16 = 4$ and $160/16 = 10$ groups. Counting one stem convolution, three convolutions per block, and the linear head gives 32 layers, hence the name.

```
class RegNetX32(AnyNet):
    def __init__(self, lr=0.1, num_classes=10, num_steps=None):
        stem_channels, group_width, bot_mul = 32, 16, 1
        depths, channels = (4, 6), (64, 160)
        super().__init__(  # The number of groups is c_i / g
            tuple((d, c, c // group_width, bot_mul)
                  for d, c in zip(depths, channels)),
            stem_channels, lr, num_classes)
        self.num_steps = num_steps  # Length of the learning-rate schedule
```

The layer summary shows the resolution falling from 96 to 48 in the stem and then to 24 and 12 in the two stages, while the channels widen.

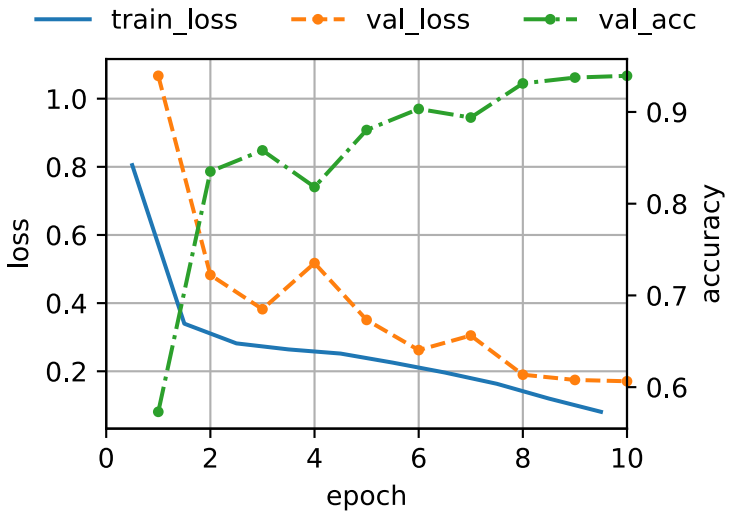
```
RegNetX32().layer_summary((1, 1, 96, 96))
```

```
Sequential              (1, 32, 48, 48)
Sequential              (1, 64, 24, 24)
Sequential              (1, 160, 12, 12)
Sequential              (1, 10)
```

Counted as for AlexNet in Section 7.1, RegNetX32 performs 112 million multiply-adds per 96×96 image. ResNet-18 performs 319 million, almost three times as many.

How well does a network built from these rules do against ResNet-18? We train on Fashion-MNIST at 96×96 with minibatches of 128 for 10 epochs, with the optimizer of `AnyNet` at a peak rate of 0.1, so that the warmup takes the first epoch.

```
data = d2l.FashionMNIST(batch_size=128, resize=(96, 96))
model = RegNetX32(lr=0.1, num_steps=10 * len(data.train_dataloader()))
trainer = d2l.Trainer(max_epochs=10, num_gpus=1)
trainer.fit(model, data)
```



We count the parameters of the trained network and read off its final validation accuracy.

```
print(f'{d2l.num_params(model):,} parameters, validation accuracy '
      f'{trainer.history["val_acc"][-1]:.3f}')
```

```
519,658 parameters, validation accuracy 0.939
```

For a fair comparison, ResNet-18 must be trained with the same optimizer. Conveniently, it fits the AnyNet template. We override `stem` and `stage` with the stem and the residual block `d2l.Residual` of Section 7.4, which gives the network of that section, and it inherits the op-

timizer that we attached to `AnyNet`. Each entry of `arch` now holds a depth, a width, and the stride of the stage's first block.

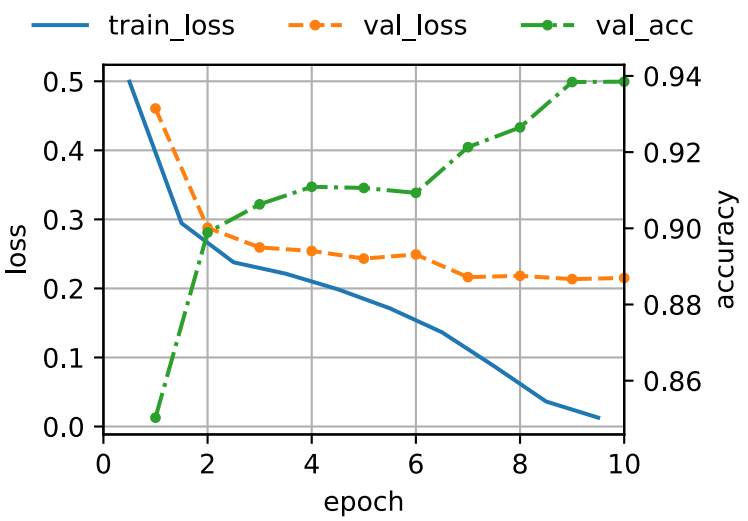
```
class ResNet18(AnyNet):
    def __init__(self, lr=0.1, num_classes=10, num_steps=None):
        super().__init__(((2, 64, 1), (2, 128, 2), (2, 256, 2), (2, 512, 2)),
                         64, lr, num_classes)
        self.num_steps = num_steps

    def stem(self, num_channels):
        return nn.Sequential(
            nn.LazyConv2d(num_channels, kernel_size=7, stride=2, padding=3,
                          bias=False),
            nn.LazyBatchNorm2d(), nn.ReLU(),
            nn.MaxPool2d(kernel_size=3, stride=2, padding=1))

    def stage(self, depth, num_channels, stride):
        blk = [d2l.Residual(num_channels, use_1x1conv=stride > 1,
                            strides=stride)]
        blk += [d2l.Residual(num_channels) for _ in range(depth - 1)]
        return nn.Sequential(*blk)
```

We train it on the same data for the same 10 epochs.

```
model = ResNet18(lr=0.1, num_steps=10 * len(data.train_dataloader()))
trainer = d2l.Trainer(max_epochs=10, num_gpus=1)
trainer.fit(model, data)
```




```
print(f'{d2l.num_params(model):,} parameters, validation accuracy '
      f'{trainer.history["val_acc"][-1]:.3f}')
```

```
11,175,370 parameters, validation accuracy 0.939
```

ResNet-18 has 11,175,370 parameters and RegNetX32 519,658, less than 5% as many, and both reach about 94% validation accuracy after 10 epochs. Over three or more runs in each framework, the accuracies of the two networks spread over the same range, from about 93.7% to 94.2%, so the difference between the two printed accuracies is run-to-run noise. What does the tie show? It cannot show that the rules produce better networks. The two populations above tested one rule with everything else drawn alike; these two networks differ in depth, width, and block all at once, and 94% will turn out to be the ceiling of this dataset. It does show that ResNet-18 is far larger than the task requires. A network that obeys the rules reaches the same accuracy with about a third of the multiply-adds.

The optimizer is part of the result. With the plain SGD of Section 7.4, the same RegNetX32 ends near 91%, while ResNet-18 reaches 93%. The rules were found and tested with momentum and weight decay, and that is the setting in which RegNetX32 matches ResNet-18; the exercises ask which of the two ingredients accounts for the gap.

Larger members of the design space gain nothing on a dataset this easy. The paper's four-stage RegNetX-400MF, -800MF, and -1.6GF have 4.8 to 8.3 million parameters. Trained with the same recipe, and with the last batch-normalization scale of every block initialized to zero so that each block starts as the identity (Section 7.4.2), without which some runs of these deeper networks start badly, they end between 93% and 94%. At this scale the dataset, not the architecture, limits the accuracy to about 94%.

Fewer multiply-adds do not make RegNetX32 faster to train, either: on a GPU, an epoch takes about one and a half times as long as one of ResNet-18. Multiply-adds count arithmetic, not time (Section 7.5), and GPU libraries run grouped convolutions, like depthwise ones, well below the arithmetic rate of dense ones. The paper's claim of speed is accordingly a narrow one. Radosavovic et al. (2020) report that RegNets match or beat EfficientNets of equal multiply-adds in accuracy and run up to five times faster on GPUs; against ResNets they find similar GPU times only at equal numbers of activations, the values that all convolutions output for one image.

7.8.4 Summary

Hand-designed networks fix their stage depths and widths by intuition, and a single good network does not reveal which of its choices mattered. A design space is a population of networks defined by a template, here AnyNet: a stem, four stages of ResNeXt blocks with a free depth, width, group width, and bottleneck ratio each, and a head. Comparing the error distributions of briefly trained samples shows, on ImageNet, that one bottleneck ratio and one group width for all stages cost nothing, while widths and depths that grow across stages help. On Fashion-MNIST, two populations of a dozen networks each barely separate: there the width rule is worth less than a point of mean error instead of twenty. RegNet adds a bottleneck ratio of 1 and widths that grow linearly with the block index, which turns 16 free numbers into a few. On Fashion-MNIST, a 32-layer RegNetX matches the accuracy of ResNet-18, about 94%, with less than 5% of its parameters and a third of its multiply-adds, when both are trained with momentum and weight decay. Whether the rules hold for other blocks, such as those of ConvNeXt or of the vision transformers of Chapter 11, is a question the same method can answer (problem 8).

This section ends the chapter, and its experiment says something about the others. On upsampled Fashion-MNIST, ResNet-18 already sits at about 94%, which leaves the designs that followed it little room to show an advantage in accuracy; for that we had to quote ImageNet results. What the chapter does leave us with is a short list of parts that survived every

redesign (blocks repeated in stages, normalization, residual connections) and the habit of asking whether a gain belongs to the network or to the way it was trained.

7.8.5 Exercises

1. code **Four stages at matched budget.** `RegNetX32` uses two stages. Extend it to the four stages of Figure 7.21, keeping widths and depths nondecreasing across stages and the parameter count within 10% of `RegNetX32`. Compare validation accuracy and time per epoch on Fashion-MNIST at 96×96. Work out the feature-map size at which each stage operates and explain what a 3×3 convolution computes on the last one.

2. **Quantized linear widths.** Radosavovic et al. (2020) turn the observation that width should grow linearly with the block index into a rule with three numbers. For d blocks, the target width of block j is the linear rule of Section 7.8.3,

$$u_j = u_0 + a\, j, \qquad j = 0, \dots, d-1. \tag{7.14}$$

   Each block is snapped to a geometric grid with ratio m,

$$s_j = \text{round}\left(\frac{\log(u_j/u_0)}{\log m}\right), \qquad \tilde{u}_j = u_0\, m^{s_j}. \tag{7.15}$$

   Blocks with equal $\tilde{u}_j$ form one stage.

   1. For $d = 13$, $u_0 = 24$, $a = 36.44$, and $m = 2.49$ (the paper's w_0, w_a, and w_m), compute the number of stages, the depth of each, and the stage widths rounded to the nearest multiple of 8. Compare with the RegNetX-200MF configuration in the paper.
   2. Describe how the number of stages, and how closely $\tilde{u}_j$ tracks u_j, change as $m \to 1$ and as m grows large.
   3. A shared group width must divide every stage width. Which of your widths violate this for a group width of 16, and how would you adjust them?

3. code **RegNetY.** RegNetY adds the squeeze-and-excitation gate `SE` of Section 7.5.2.3 to every block. Insert `SE` into `d2l.ResNeXtBlock` after its grouped 3×3 convolution, as Radosavovic et al. (2020) do, and build a RegNetY32 with the configuration of `RegNetX32`.

   1. Predict the added parameter count from the estimate of about $2c^2/r$ per gate and verify it against the model.
   2. Compare validation accuracy and time per epoch with `RegNetX32`. Where does the extra time come from, given that the gate adds almost no multiply-adds?

4. **Design spaces versus architecture search.** Neural architecture search (Zoph and Le 2016; Liu et al. 2018) selects one network from a search space; the method of Section 7.8.2 constrains a distribution over networks.

   1. Explain why comparing empirical CDFs Equation 7.13 cannot identify the best network in a space, and why that is not its purpose.
   2. Of the four assumptions listed in Section 7.8.2, which does a NAS method that trains candidates briefly also rely on, and which are specific to the design-space approach?
   3. Name one failure mode of each approach: what can go wrong when a single architecture is selected by a proxy score, and what can go wrong when a design space is constrained by rules fitted to a sample?

5. code **Breaking the rules.** The two populations of this section tested the rule on widths. Implement variants of `RegNetX32` that each violate one of the other design principles of AnyNetX$_E$ in Section 7.8.3, at roughly matched parameter count: depths that decrease across stages, group widths that differ between stages, and bottleneck ratios that differ between stages.
    1. Report validation accuracy for each violation against `RegNetX32`. Train every network three times: runs of one network differ by a few tenths of a point.
    2. Which of depth d_i, group width g_i, and bottleneck ratio b_i matters most here?
    3. Compare your ranking with Figure 7.22. Which violations did the sampled design spaces of Radosavovic et al. (2020) also find harmless, and why does one network per violation give weaker evidence than an error CDF?
6. code **The optimizer.** With plain SGD at a peak rate of 0.1, `RegNetX32` ends several points below the accuracy it reaches with momentum and weight decay.
    1. Train it with momentum 0.9 and no weight decay, and with plain SGD at peaks of 0.1 and 1.0. Which change accounts for most of the gain?
    2. Show that when the gradient stays constant, SGD with momentum β at rate η eventually takes the same steps as plain SGD at rate $\eta/(1-\beta)$. Do your runs agree?
    3. Repeat the comparison for `ResNet18`. Does it gain as much from the change of optimizer?
7. [extended] **Proxy reliability.** The method of Section 7.8.2 assumes that accuracy after a few epochs predicts final accuracy and that results at small scale carry over to larger networks. Test both with eight configurations sampled from a two-stage AnyNet space with $d_i \in \{2, 4, 6\}$, $c_i \in \{16, 32, 64, 128\}$ subject to $c_1 \leq c_2$, a number of groups in $\{4, 8\}$ shared by both stages (the `groups` argument of `d2l.ResNeXtBlock`), and $b = 1$. The *rank correlation* of two rankings is the correlation between the positions the networks occupy in them.
    1. Train each configuration for 10 epochs on Fashion-MNIST at 96×96, recording validation accuracy after every epoch. Compute the rank correlation between the ranking after 1, 2, and 5 epochs and the ranking after 10 epochs.
    2. Repeat the eight runs with every width halved and the input resized to 48×48. Compute the rank correlation between the final rankings at small and at full scale.
    3. Which of the two assumptions holds up better in your experiment? What would a failure of the weaker one do to conclusions drawn from short, small-scale runs such as those behind Figure 7.22?
8. [extended] **A ConvNeXt design space.** The design spaces of Figure 7.22 were built from ResNeXt blocks. Build a two-stage AnyNet space from the `ConvNeXtBlock` of Section 7.7 instead, with the stem and downsampling layer of `ConvNeXt`. Stage i has depth $d_i \in \{1, 2, 4\}$, width $c_i \in \{32, 64, 128\}$, and an expansion ratio in $\{2, 4\}$. `ConvNeXtBlock` fixes the expansion ratio at 4, so add it as an argument.
    1. Which of the AnyNet parameters d_i, c_i, g_i, and b_i have a counterpart in this space, and which does the block fix?
    2. Sample 16 networks from the unconstrained space, 16 with one expansion ratio shared by both stages, and 16 with $c_1 \leq c_2$ and $d_1 \leq d_2$. Train each for two epochs on Fashion-MNIST at 96×96 and plot the three empirical CDFs Equa-

tion 7.13. Which differences are resolved at 16 samples per space?
3. Which of the RegNet design principles survive the change of block?

Chapter 8

Sequence Models

After two chapters on images, the last chapter of this part turns to sequences. Every model so far reads an input of fixed shape, such as a 28×28 image, and treats each example on its own. A sentence, a recording, or a series of measurements is different: it varies in length, and its entries depend on each other and on their order. Take the opening sentence of H. G. Wells' *The Time Machine*, which the models of this chapter read: "The Time Traveller (for so it will be convenient to speak of him) was expounding a recondite matter to us." To guess its last word, a reader uses the nineteen words before it, in order. How can a model do the same when the text before it can have any length?

Claude Shannon made a game of this. In 1951 he measured how predictable English is by asking people to guess a text one letter at a time. A *language model* plays the same game: it predicts each token, which may be a character or a word, from the tokens before it. A model that can predict can also write, by feeding each token it chooses back in as its next input.

The difficulty is the text already read, which grows with every token. A window of the last few tokens forgets everything older. A *recurrent neural network* instead summarizes all of it in a *hidden state* of fixed size, which the same layer updates after every token. Today's large language models also predict one token at a time, so what this chapter teaches about cutting text into tokens, scoring a model, and turning its predictions into text carries over to them.

The chapter sets up the problem before any recurrent network appears. We predict a sequence from its own past and see why forecasting many steps ahead is harder than forecasting one (Section 8.1), cut text into tokens (Section 8.2), and build language models that only count, which also gives us the measure by which language models are compared (Section 8.3). The counts run out, and the recurrent network is our answer. Section 8.4 replaces the window by a hidden state, and Section 8.5 implements and trains the model. We then put it to the test: against the counts, without the safeguard that its training turns out to need, and position by position, to see how much of what it has read the model uses. Section 8.6 explains what we find from the gradients of that same model. The last section, Section 8.7, turns next-token distributions into text.

Gated cells and state space models, which let a recurrent network remember further back, follow in Chapter 12, and attention, which keeps every past token, in Chapter 10.

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.

Books

- Speech and Language Processing, 3rd ed. — Jurafsky & Martin — free draft (complete PDF, updated January 2026); Chapter 2 (words and tokens) and Chapter 3 (n-gram language models) are the standard treatment of Section 8.2 and Section 8.3, and treat at length the interpolated and discounted smoothing schemes, of which Section 8.3 derives Kneser–Ney; Chapter 13 covers RNNs and LSTMs.
- Deep Learning — Goodfellow, Bengio & Courville — free HTML; Chapter 10 (Sequence Modeling) develops recurrent networks, teacher forcing, and backpropagation through time with fuller gradient derivations than Section 8.4 and Section 8.6, and introduces the gated architectures of Chapter 12.
- Natural Language Processing — Jacob Eisenstein — free PDF of the notes that became *Introduction to Natural Language Processing* (MIT Press, 2019); Chapter 6 treats n-gram and RNN language models as one story, the step this chapter takes from Section 8.3 to Section 8.4.

Courses and video lectures

- Stanford CS224N: Natural Language Processing with Deep Learning — free slides and notes, with complete lecture videos on YouTube; its language-modeling and RNN lectures cover the material from Section 8.3 to Section 8.6, the vanishing-gradient argument included, before turning to transformers.
- Neural Networks: Zero to Hero — Andrej Karpathy — free video series; the *makemore* lectures build character-level language models from a counting bigram model up to neural networks, the path from Section 8.3 to Section 8.5 typed live, and "Let's build the GPT Tokenizer" builds the byte-level BPE of Section 8.2.
- MIT 6.S191: Introduction to Deep Learning — Amini & Amini — free, updated annually; the "Deep Sequence Modeling" lecture is a fast, visual pass over sequence tasks, RNNs, and their gradient pathologies (Section 8.4, Section 8.6).

Foundational papers

- Prediction and Entropy of Printed English — Claude E. Shannon (1951), *Bell System Technical Journal* — free scan; frames language as a game of guessing the next symbol and measures the entropy of English with human predictors, the origin of the perplexity and bits per byte of Section 8.3.
- Finding Structure in Time — Jeffrey Elman (1990), *Cognitive Science* — paywalled, widely reproduced online; introduces the network of Section 8.4, whose hidden state, fed back on itself, discovers word boundaries and grammatical categories from raw sequences.
- A Neural Probabilistic Language Model — Bengio, Ducharme, Vincent & Jauvin (2003), *JMLR* — free; replaces the sparse count tables of Section 8.3 by learned embeddings and a neural network, the neural n-gram of Section 8.4 and the embedding and softmax layers of Section 8.5.
- A Hierarchical Bayesian Language Model Based on Pitman–Yor Processes — Yee Whye Teh (2006), *ACL* — free; gives the Kneser–Ney smoothing of Section 8.3 a Bayesian reading, as approximate inference in a hierarchical Pitman–Yor process, a model that the sequence memoizer of Wood et al. (2011) extends to contexts of unbounded length.

- Recurrent Neural Network Based Language Model — Mikolov et al. (2010), *Interspeech* — free; shows that RNN language models beat smoothed n-grams in practice, with the model of Section 8.5 evaluated by perplexity as in Section 8.3.
- On the Difficulty of Training Recurrent Neural Networks — Pascanu, Mikolov & Bengio (2013), *ICML* — free; the analysis of vanishing and exploding gradients in Section 8.6 and the source of the gradient-norm clipping of Section 8.5. The difficulty itself was first formalized by Bengio, Simard & Frasconi (1994) (paywalled).
- Neural Machine Translation of Rare Words with Subword Units — Sennrich, Haddow & Birch (2016), *ACL* — free; brought byte pair encoding into NLP, the merge algorithm implemented from scratch in Section 8.2.
- Language Models are Unsupervised Multitask Learners — Radford et al. (2019) — free; the GPT-2 report, source of the byte-level BPE with regex pre-tokenization that Section 8.2 rebuilds and checks token for token against `tiktoken`, and of the top-k sampling of Section 8.7.
- The Curious Case of Neural Text Degeneration — Holtzman et al. (2020), *ICLR* — free; explains why maximization loops and why the unreliable tail spoils pure sampling, and proposes nucleus (top-p) sampling, the argument Section 8.7 follows.
- Turning Up the Heat: Min-p Sampling for Creative and Coherent LLM Outputs — Nguyen et al. (2025), *ICLR* — free; the min-p rule in the sampler of Section 8.7, which keeps tokens in proportion to the model's confidence, with evidence that it degrades less than top-p at high temperatures.

Tutorials, notes, and interactive

- The Unreasonable Effectiveness of Recurrent Neural Networks — Andrej Karpathy — free; the 2015 essay in which character-level RNN language models write Shakespeare, Wikipedia markup, LaTeX, and C code, and single hidden units track quotes and line lengths; a larger version of what Section 8.5 trains, sampled as in Section 8.7.
- minbpe — Andrej Karpathy — free; a minimal, commented byte-level BPE codebase (the companion to the tokenizer video above) that reproduces GPT-4's `tiktoken` tokenizer, a second implementation to compare with the one of Section 8.2.
- Tiktokenizer — free, nothing to install; shows how production tokenizers segment text as it is typed, a quick way to repeat the fertility and digit experiments of Section 8.2.5.3 on GPT-2 and current vocabularies.
- How to Generate Text — Patrick von Platen (Hugging Face) — free; greedy search, beam search, top-k, and top-p with runnable `transformers` code, the experiments of Section 8.7 repeated on models large enough to show each failure clearly.
- Hugging Face LLM Course, Chapter 6: Tokenizers — free; trains and dissects production BPE, WordPiece, and Unigram tokenizers with the `tokenizers` library, a companion to the production tokenizers examined at the end of Section 8.2.

8.1 Working with Sequences

An email client that sees "Thank you for your" proposes "help" or "time". A weather service that has recorded the temperature every hour forecasts the next hour. In both cases the best prediction depends on what came before, and on its order: "your thank for you" calls for a different continuation, and a falling temperature curve for a different forecast than a rising one. The models of the earlier chapters cannot express this. They map one input of fixed shape, a vector $\mathbf{x} \in \mathbb{R}^d$, to one output, and they treat examples as independent. A sentence, a recording, or a series of measurements has neither property: its length varies,

and its entries depend on each other.

This section turns sequence modeling into a problem we already know how to solve, regression. We predict each entry from the entries before it, and we keep the input of the predictor at a fixed size: either a window of recent entries or a running summary of everything seen so far. We then fit a linear model on a fixed window to a noisy sine wave, where we know the truth. Predicting one step ahead turns out to be easy. Predicting many steps ahead, by feeding the model its own predictions, is hard. The rest of the chapter applies the same ideas to text.

8.1.1 Sequential Data

A *sequence* is an ordered list of feature vectors $\mathbf{x}_1, \dots, \mathbf{x}_T$, or of scalars $x_1, \dots, x_T$. The index $t \in \mathbb{Z}^+$ is a *time step*, and the length T may differ from one sequence to the next. Text is a sequence of characters or words. Audio, whether speech or music, is a sequence of samples, or of the spectrogram frames computed from them. Time series such as prices, sensor readings, and patient vitals arrive one measurement at a time.

We still assume that *entire* sequences (a whole document, a whole patient trajectory) are drawn independently from one fixed distribution over sequences. *Within* a sequence, however, the entries depend on each other: the medicine a patient receives on day ten of a hospital stay depends heavily on the previous nine days. Far from being a nuisance, this dependence is what makes the past useful: were the entries independent, conditioning on earlier ones would not improve a prediction of the next. We do not even assume that the entries share one distribution. A document reads differently at its end than at its beginning, and a patient's status evolves toward recovery or decline.

Prediction tasks over sequences come in a few shapes. Sometimes the target is a single label for an entire input sequence (the sentiment of a movie review), sometimes a sequence given a fixed input (captioning an image), and sometimes a sequence given a sequence (translating a sentence, transcribing speech). Before handling targets of any kind, we model the sequence itself: we estimate how probable a given sequence is, $P(x_1, \dots, x_T)$. Why would anyone want this number? Estimating it needs no labels, since every entry serves as the target for the entries before it, and a model that knows which sequences are probable can generate new ones and choose between candidates, say two transcriptions of one recording.

8.1.2 Autoregressive Models

Take a numerical series such as a stock index or a temperature reading. A forecaster who wants to act on the next step cares about the conditional distribution of that step given the observations before it,

$$P(x_t \mid x_1, \dots, x_{t-1}). \tag{8.1}$$

The history $x_1, \dots, x_{t-1}$ appears in almost every formula of this chapter, so we abbreviate it as $x_{<t}$. Even when the full distribution is hard to pin down, a useful summary is the conditional expectation $\mathbb{E}[x_t \mid x_{<t}]$, which we could try to fit with linear regression (Section 2.1). A model that regresses a signal on its own past is an *autoregressive model*: features and target come from the same series.

There is an immediate obstacle. The number of conditioning inputs in $x_{<t}$ grows with t, so if we treat the history as a training set, every example has a different number of features

and no fixed-input model applies. Two strategies produce fixed-size representations of the history (Figure 8.1). Both are autoregressive. They differ in what they keep of the past.

8.1.2.1 Fixed Windows

The first strategy conditions only on the last m observations $x_{t-m}, \dots, x_{t-1}$ and discards older history. Every example then has m features (for $t > m$), so any model with a fixed input length, linear or deep, applies. This is the classical form of autoregression, and m is the *window length*. A window is simple, and since every prediction depends only on its own window, all of them can be computed at once. Its weakness is a hard horizon: anything older than m steps is invisible to the model.

8.1.2.2 Latent Summaries

The second strategy keeps a running summary of the observations $x_1, \dots, x_t$, a vector $\mathbf{h}_t$, and updates it as each new observation arrives, $\mathbf{h}_t = g(\mathbf{h}_{t-1}, x_t)$. We predict the next entry from the summary, $\hat{x}_{t+1} = f(\mathbf{h}_t)$. Because $\mathbf{h}_t$ is never observed directly, this is a *latent autoregressive model*. The state $\mathbf{h}_t$ has the same size however long the history grows, so a past of any length fits into bounded memory. Recurrent neural networks (Section 8.4) are latent autoregressive models with a learned g, and so are the state space models of Chapter 12. What the state should retain, and how to update it, are the questions of those sections.

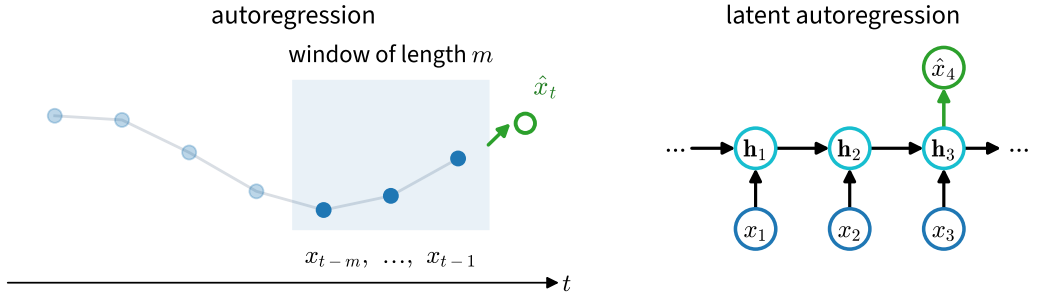


Figure 8.1: Two ways to give a sequence model a fixed-size input. (left) A fixed window conditions on the last m observations and discards the rest. (right) A latent summary carries a state $\mathbf{h}_t$ that summarizes the whole past in fixed size.

8.1.2.3 From Conditionals to Sequences

Whichever strategy we pick, the one-step conditionals determine a model of the whole sequence. The *chain rule of probability* writes any joint distribution as a product of conditionals, one factor per entry, each conditioned on all the entries before it:

$$P(x_1, \dots, x_T) = \prod_{t=1}^{T} P(x_t \mid x_{<t}). \tag{8.2}$$

The first factor has nothing to condition on, since $x_{<1}$ is empty, and is $P(x_1)$. The factorization is exact: it holds for every distribution over sequences. Thus a model of the next entry is a model of the entire sequence. For three words,

$$P(\text{the}, \text{cat}, \text{sat}) = P(\text{the})\, P(\text{cat} \mid \text{the})\, P(\text{sat} \mid \text{the}, \text{cat}),$$

and a model that assigns 0.05, 0.01, and 0.2 to the three factors gives the sentence probability 10^{-4}. To score a sequence, we multiply its conditionals, or add their logarithms, which avoids underflow for long sequences. To generate a sequence, we sample x_1, then x_2 given x_1, and so on, feeding each sample into the next condition.

The chain rule leaves us one choice, and we made it without saying so: we factored the sequence left to right. Any order gives a valid factorization, but for causal data the forward direction is usually easier to learn, since predicting an effect from its cause tends to be simpler than the reverse (J. Peters et al. 2017). We want to predict the future from the past anyway, so we factor left to right throughout.

When the entries are discrete tokens, each conditional is a classifier over the vocabulary, and estimating $P(x_1, \dots, x_T)$ is called *language modeling* (Section 8.3). For now we keep the values continuous and focus on prediction.

8.1.3 Markov Models and Stationarity

A fixed window assumes that the observations it discards do not matter, and fitting one model to every position assumes that the series behaves alike everywhere. Both assumptions have names.

8.1.3.1 The Markov Condition

When conditioning on the last m steps loses no predictive power relative to the full history, we say the sequence satisfies a *Markov condition*: the future is conditionally independent of the distant past given the recent past. With $m = 1$ the series follows a *first-order* Markov model, and in general an m^{th}-order one. Under the first-order condition each factor of Equation 8.2 depends on its predecessor only, and the factorization collapses to a product over adjacent pairs:

$$P(x_1, \dots, x_T) = P(x_1) \prod_{t=2}^{T} P(x_t \mid x_{t-1}). \tag{8.3}$$

In practice the Markov condition rarely holds exactly. Real text keeps yielding information as we widen the context, though each additional token adds less than the one before. We often assume an m^{th}-order condition anyway, because it keeps the model small enough to compute and to estimate from data. Even large language models read a context window of fixed length, not everything that came before it. For discrete data the simplest such model counts: $P(x_t \mid x_{t-1})$ is estimated by the relative frequency of each successor, an idea we take up again for n-grams in Section 8.3.

8.1.3.2 Stationarity

Fitting any of these models assumes that the *rule* generating each entry from its predecessors does not itself change over time. Dynamics with this property are called *time-homogeneous* or, more loosely and far more often, *stationary*. The values still change, sometimes dramatically; what stays fixed is the conditional distribution $P(x_t \mid x_{t-m}, \dots, x_{t-1})$.

A fixed rule does not make the entries identically distributed: a document still reads differently at its end, because the rule meets a different context there. It is this assumption that lets us pool examples from different parts of one sequence, or from many sequences, into a single training set, and it is why we build training data by sliding a window along a series.

8.1.4 Forecasting a Noisy Sine Wave

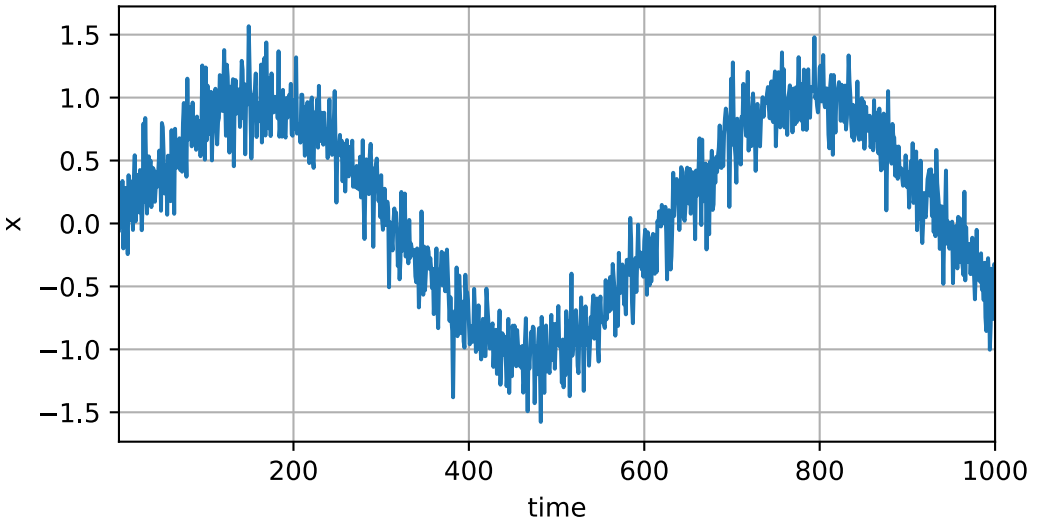
Let us try this out on data where we control the ground truth. We fit an autoregressive model and compare predicting one step ahead with predicting many steps ahead.

8.1.4.1 Data and Model

The series has $T = 1000$ entries $x_t = \sin(0.01\, t) + \epsilon_t$, where the noise ϵ_t is drawn from $\mathcal{N}(0, 0.2^2)$ independently at every step. One period of the sine spans $2\pi/0.01 \approx 628$ steps.

```
class Data(d2l.DataModule):
    def __init__(self, batch_size=16, T=1000, num_train=600, m=4):
        self.save_hyperparameters()
        self.time = d2l.arange(1, T + 1, dtype=d2l.float32)
        self.x = d2l.sin(0.01 * self.time) + d2l.randn(T) * 0.2
```

```
data = Data()
d2l.plot(data.time, data.x, 'time', 'x', xlim=[1, 1000], figsize=(6, 3))
```



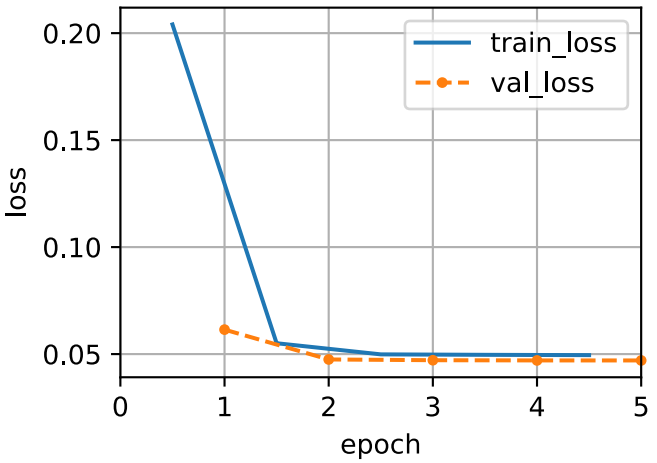
To turn the series into supervised examples, we adopt the fixed-window strategy with window length m: the label is $y = x_t$ and the features are $(x_{t-m}, \ldots, x_{t-1})$. This yields $T - m$ examples, since the first m steps lack enough history and we drop them (padding with zeros is the alternative). The method `lag_features` builds all of them at once, one column per lag, and the data loader serves the first 600 for training, a bit less than one period of the sine.

```
@d2l.add_to_class(Data)
def lag_features(self):
    """Features (x_{t-m}, ..., x_{t-1}) and labels x_t for t > m."""
    features = [self.x[i : self.T-self.m+i] for i in range(self.m)]
    return d2l.stack(features, 1), d2l.reshape(self.x[self.m:], (-1, 1))

@d2l.add_to_class(Data)
def get_dataloader(self, train):
    i = slice(0, self.num_train) if train else slice(self.num_train, None)
    return self.get_tensorloader(self.lag_features(), train, i)
```

Our model is linear regression on the $m = 4$ lag features, about the simplest autoregressive model there is.

```
model = d2l.LinearRegression(lr=0.01)
trainer = d2l.Trainer(max_epochs=5)
trainer.fit(model, data)
```

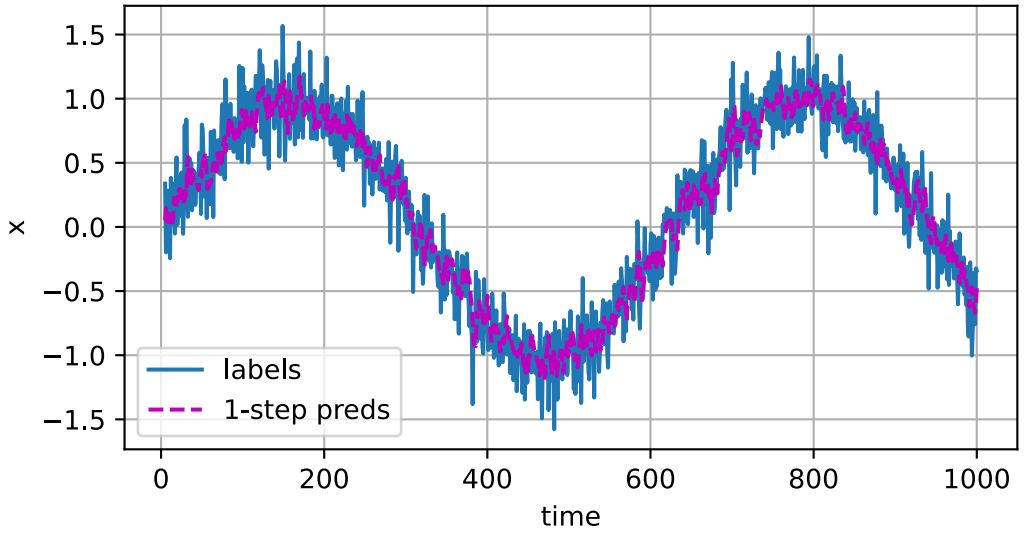


8.1.4.2 One-Step-Ahead Prediction

First the easy case: predict $\hat{x}_t$ from the *observed* previous m values, for every $t > m$.

```
features, labels = data.lag_features()
with torch.inference_mode():  # prediction only: record no gradients
    onestep_preds = d2l.numpy(model(features))
```

```
d2l.plot(data.time[data.m:], [labels, onestep_preds], 'time', 'x',
         legend=['labels', '1-step preds'], figsize=(6, 3))
```



The predictions track the series closely, also beyond step 600, where the model sees data it was not trained on.

8.1.4.3 Multistep Rollout

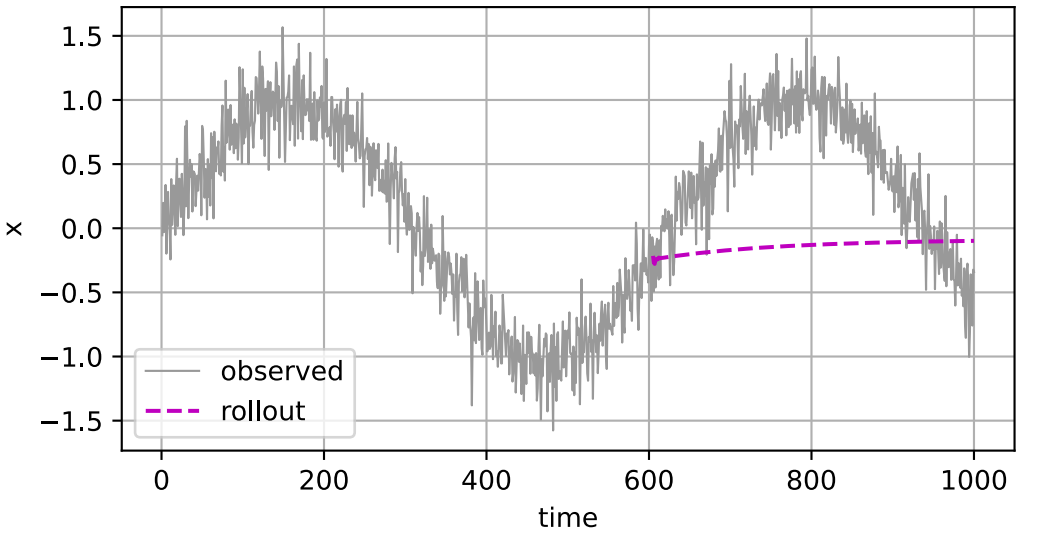
The hard case is forecasting several steps ahead. Suppose that the observations stop at some step t and that we want to predict $\hat{x}_{t+5}$. We cannot form the one-step input for it, because we have not observed $x_{t+1}, \ldots, x_{t+4}$. The fix is to feed our own predictions back in as if they were observations, rolling forward one step at a time:

$$\begin{aligned}
\hat{x}_{t+1} &= f(x_{t-3}, x_{t-2}, x_{t-1}, x_t),\\
\hat{x}_{t+2} &= f(x_{t-2}, x_{t-1}, x_t, \hat{x}_{t+1}),\\
\hat{x}_{t+3} &= f(x_{t-1}, x_t, \hat{x}_{t+1}, \hat{x}_{t+2}),\\
\hat{x}_{t+4} &= f(x_t, \hat{x}_{t+1}, \hat{x}_{t+2}, \hat{x}_{t+3}),\\
\hat{x}_{t+5} &= f(\hat{x}_{t+1}, \hat{x}_{t+2}, \hat{x}_{t+3}, \hat{x}_{t+4}).
\end{aligned}$$

From the fifth step on, the model sees only its own predictions. The prediction $\hat{x}_{t+k}$ is the *k-step-ahead prediction*, and a *rollout* chains one-step predictions to reach it. The loop below rolls forward from $t = 604$, the last value that the training examples use, to the end of the series.

```
multistep_preds = data.x.clone()
with torch.inference_mode():
    for i in range(data.num_train + data.m, data.T):
        multistep_preds[i] = model(
            multistep_preds[i-data.m : i].reshape(1, -1))
multistep_preds = d2l.numpy(multistep_preds)
```

```
start = data.num_train + data.m
d2l.plot([data.time, data.time[start:]], [data.x, multistep_preds[start:]],
         'time', 'x', figsize=(6, 3))
ax = d2l.plt.gca()
ax.lines[0].set(color='0.6', linewidth=0.8)  # The series: thin and grey
ax.legend(['observed', 'rollout']);
```



The rollout never follows the oscillation. It starts at the level of the last observed window and drifts slowly toward a constant, while the series rises and falls. How can a model with so small a one-step error fail so thoroughly? The learned weights tell us.

```
w, b = model.get_w_b()
print(f'weights on x[t-{data.m}], ..., x[t-1]:',
      d2l.numpy(d2l.reshape(w, -1)).round(3))
```

```
weights on x[t-4], ..., x[t-1]: [0.25  0.249 0.23  0.254]
```

All four weights are close to $1/4$: the model has learned to predict the next value as the average of the last four. For one step this is a sound rule. The sine changes by at most 0.01 from one step to the next, while the noise has a standard deviation of 0.2. An average of four noisy observations carries only a quarter of their noise variance, $0.04/4 = 0.01$, and lagging the curve by a step or two adds almost no error. Fed its own outputs, however, an average of the last four values can only repeat their current level. Because the weights sum to slightly less than one, that level also drifts slowly toward the rollout's fixed point $b/(1 - \sum_i w_i)$, where b is the bias: a small bias divided by a small difference, so the fixed point differs from run to run. The model has not learned the curvature of the sine, which the one-step loss barely rewarded, and over many steps that omission dominates the error. Linear models are not to blame. Without noise, the sine satisfies $x_t = 2\cos(0.01)\, x_{t-1} - x_{t-2}$ exactly, so two lags with weights of nearly 2 and -1 would continue it forever. On the noisy series those weights would multiply the noise variance by five, and the fit prefers to average (exercise 1).

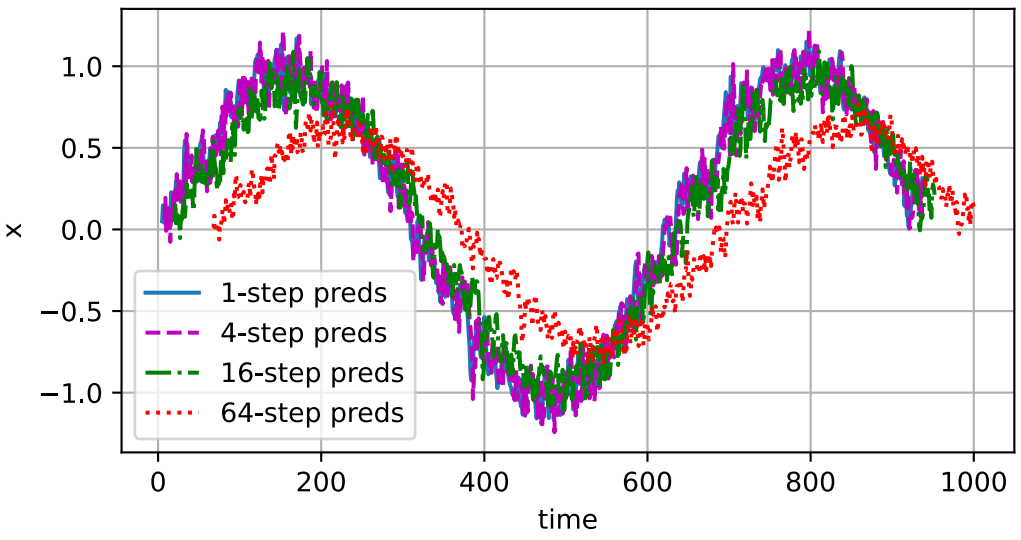
8.1.4.4 Error Growth with the Horizon

How fast does the error grow with the horizon k? To find out, we compute k-step-ahead predictions from every start position at once. `k_step_pred` keeps a list of columns, each with one entry per start position: first the m observed lags, then, round by round, the predictions made from the last m columns. Entry $j - 1$ of the returned list holds the j-step-ahead predictions.

```
def k_step_pred(model, x, m, k):
    """Predictions 1, 2, ..., k steps ahead from every start position."""
    n = len(x) - m - k + 1                      # number of start positions
    columns = [x[i : i + n] for i in range(m)]   # observed lags
    for j in range(k):
        preds = model(d2l.stack(columns[j : j + m], 1))
        columns.append(d2l.reshape(preds, -1))
    return columns[m:]                          # entry j-1: j steps ahead
```

```
steps = (1, 4, 16, 64)
with torch.inference_mode():
    preds = k_step_pred(model, data.x, data.m, steps[-1])
```

```
d2l.plot([data.time[data.m+k-1 : data.T-steps[-1]+k] for k in steps],
         [d2l.numpy(preds[k-1]) for k in steps], 'time', 'x',
         legend=[f'{k}-step preds' for k in steps], figsize=(6, 3))
```



The 1- and 4-step forecasts follow the series. The 16-step forecast lags it visibly, and the 64-step forecast is a copy of the series delayed by about 64 steps and visibly shrunk in amplitude; how much it shrinks depends on how far the weights' sum falls short of one. The mean squared error against the observed values puts numbers on this. As a reference, the variance of the series is the mean squared error of a forecast that always predicts the mean.

```
x = d2l.numpy(data.x)
for k in steps:
    err = d2l.numpy(preds[k-1]) - x[data.m+k-1 : data.T-steps[-1]+k]
    print(f'{k:2d} steps ahead: mean squared error {(err**2).mean():.3f}')
print(f'variance of the series: {x.var():.3f}')
```

```
 1 steps ahead: mean squared error 0.048
 4 steps ahead: mean squared error 0.049
16 steps ahead: mean squared error 0.066
64 steps ahead: mean squared error 0.246
variance of the series: 0.496
```

One step ahead, the error is about 0.05: the noise variance $0.2^2 = 0.04$, which no forecast can remove, plus the 0.01 of noise left in an average of four observations. The growth with k follows from the delay. A k-step forecast is roughly the series k steps earlier, so its additional error is how far the sine moves in k steps. Averaged over the phase a of the sine, that squared change is

$$\frac{1}{2\pi}\int_0^{2\pi} \big(\sin(a + 0.01\,k) - \sin a\big)^2\, da = 1 - \cos(0.01\,k), \tag{8.4}$$

which is 0.001 for $k = 4$, 0.013 for $k = 16$, and 0.20 for $k = 64$. Added to the floor of 0.05, it predicts errors of about 0.05, 0.05, 0.065, and 0.25, close to the measured ones. Sixty-four steps ahead, the error is four to six times the one-step error and roughly half the variance of the series.

None of this error has accumulated. The averaging model damps whatever is fed back into it, so its rollout drifts toward a constant instead of diverging, and a k-step forecast is wrong because the sine has moved on, not because of a pile of earlier mistakes. What the experiment shows is that a model trained on a one-step loss learns only what one step rewards. Used many steps ahead, a model is as good as its dynamics, and a small one-step error does not tell us how good those are.

Rollouts have a second, better-known problem, which this experiment does not demonstrate: every prediction error becomes part of the input of all later steps, and a model that amplifies its errors instead of damping them drifts away from the data. A language model that samples one token at a time can drift off topic or into repetition once a few unusual choices steer it into text unlike anything it saw during training, and a learned simulator of a physical system can lose coherence over a long rollout in the same way. Two lines of work address it. A model can be trained on its own predictions as well as on observed histories, so that the multistep setting it is used in is no longer new to it; whether that helps our linear model is the question of exercise 7. And the rule that turns each predicted distribution into a concrete next value can avoid the choices that derail a rollout. For text, such rules are the subject of Section 8.7, at the other end of this chapter. Before we can roll out a model of text, we need text in a form that a model can read.

8.1.5 Summary

Autoregressive models turn sequence modeling into ordinary supervised learning by predicting each entry from its predecessors, either through a fixed window of the last m observations (the idea behind n-grams) or through a latent state that summarizes the whole past

in bounded memory (the idea behind recurrent networks). By the chain rule of probability, the one-step conditionals multiply to a model of the whole sequence, which can score and generate sequences. Fitting one model to all positions of a series assumes that the rule generating each entry does not change over time (stationarity). A model that predicts one step well can still be a poor model of the series, because the one-step loss rewards only what helps one step ahead. On the noisy sine wave the averaging model reaches a one-step error near the noise floor, and 64 steps ahead its error is four to six times larger. A model that amplifies its errors in a rollout would also drift away from the data; our damped model did not show this.

8.1.6 Exercises

1. code **Window length.** `Data` supplies `m=4` lag features and `d2l.LinearRegression` fits them.
    1. Increase `m` and retrain. How many past observations are needed before the validation error stops improving?
    2. Remove the noise term in `Data` and fit `m=2`. Compare the two weights with $2\cos(0.01)$ and -1, and roll the model out. Then restore the noise at standard deviations between 0.001 and 0.2. At what noise level do the weights give up the curvature and turn to averaging?
2. **Momentum strategies.** An investor selects stocks by their own past returns, expecting those that rose to keep rising. State the assumption about the return series that this strategy requires, in the terms of Section 8.1.3.1 and Section 8.1.3.2, and describe one mechanism by which the assumption fails.
3. code **Nonlinear autoregressor.** Replace `d2l.LinearRegression` by a small MLP on the same four lag features and retrain, possibly for more epochs. How do the one-step and the multistep predictions change?
4. code **Error growth with the horizon.** For each k, the last entry of the list that `k_step_pred(model, data.x, data.m, k)` returns holds the k-step-ahead predictions of `data.x[data.m+k-1:]`.
    1. For $k \in \{1, 2, 4, \dots, 256\}$, plot the mean squared k-step error against k on a logarithmic axis. Where does the curve bend upward, and how does the bend relate to the damping of the 16-step and 64-step forecasts?
    2. Find the smallest k at which the mean squared k-step error exceeds the variance of the series. What does a forecast beyond this horizon offer over the constant forecast?
5. code **Dilated and averaged lags.** ▣ Keep four features for `d2l.LinearRegression` but let them reach further back: once as the lags $x_{t-1}, x_{t-2}, x_{t-4}, x_{t-8}$, and once as the four block means $\frac{1}{4}\sum_{i=0}^{3} x_{t-4j+i}$ for $j = 1, \dots, 4$. Compare their validation errors with those for $m = 4$ and $m = 16$. Which construction helps, and why?
6. code **Regime switching.** ▣ Replace the single sine in `Data` by a series that alternates between $\sin(0.01\,t)$ and $\sin(0.05\,t)$ every 200 time steps, and keep the additive noise. Both regimes then occur among the training examples and among the validation examples. The switching series violates the stationarity assumption of Section 8.1.3.2. Does one linear model with $m = 4$ nevertheless fit both regimes, and why? Report training and validation error for each regime separately and locate the positions along the series where the fit is worst.

7. code **Scheduled sampling.** ▯ Section 8.1.4.4 mentions training a model on its own predictions as a way to narrow the gap between one-step training and multistep use. Test the idea with a variant of *scheduled sampling* (Bengio et al. 2015) for the linear model with $m = 4$. In every minibatch, replace each lag feature with probability ϵ by the model's current one-step prediction of that observation. Train one model for each $\epsilon \in \{0, 0.25, 0.5, 1\}$. For each model, measure the one-step validation error and the mean squared error of the last entry of `k_step_pred(model, data.x, data.m, 64)`. Does any $\epsilon > 0$ lower the 64-step error below its value at $\epsilon = 0$? How does the one-step error change?

8.2 From Text to Tokens

Most sequences in this chapter are text. Text poses a problem that the synthetic series of Section 8.1 did not: neural networks consume vectors of numbers, while text arrives as a string of characters. Before we can model language, we must decide how to cut a string into *tokens*, the atomic units that our models treat as observations x_t, and how to map each token to a numerical index. This choice determines the size of the model's output layer, the length of every document it reads, and even how hard arithmetic is for it. Split *The Time Machine* into characters, and a model must read some 170,000 steps. Split it into words, and the model needs an output for each of about 4,600 distinct words and still fails on any word it has not seen. Every modern language model comes with a tokenizer, and the tokenizer is best thought of as part of the model.

We first compare character and word tokens, which sit at opposite ends of a trade-off between vocabulary size and sequence length. We then represent text as raw bytes and learn a vocabulary *from data* with byte pair encoding (BPE), the subword scheme of the GPT family of models (Sennrich et al. 2016; Radford et al. 2019). Our implementation is short, and it reproduces the output of the GPT-2 tokenizer token for token.

8.2.1 Reading the Dataset

Throughout the chapter we work with H. G. Wells' The Time Machine, a novella of just over 30,000 words. Real corpora are measured in trillions of tokens, but this one is large enough to show the statistics that matter and small enough to tokenize in seconds. The `_download` method reads the raw text into a single string.

```
import collections
import re
import tiktoken
```

```
class TimeMachine(d2l.DataModule):
    """The Time Machine dataset."""
    def _download(self):
        fname = d2l.download(d2l.DATA_URL + 'timemachine.txt', self.root,
                             '090b5e7e70c295757f55df93cb0a180b9691891a')
        with open(fname) as f:
            return f.read()

data = TimeMachine()
```

```
raw_text = data._download()
raw_text[:60]
```

```
'The Time Machine, by H. G. Wells [1898]\n\n\n\n\nI\n\n\nThe Time Tra'
```

Classical preprocessing pipelines normalized text of this kind aggressively: they stripped punctuation, folded everything to lowercase, and only then split it into units. We keep such a step for the character- and word-level experiments below, and for the sections of the book that still work with such tokens. It is crude, though: normalization throws away information (capitalization distinguishes "Bill" from "bill"), and it commits us in advance to decisions about what a token may look like. The byte-level tokenizer we build later in this section needs none at all.

```
@d2l.add_to_class(TimeMachine)
def _preprocess(self, text):
    return re.sub('[^A-Za-z]+', ' ', text).lower()

text = data._preprocess(raw_text)
text[:60]
```

```
'the time machine by h g wells i the time traveller for so it'
```

8.2.2 Characters, Words, and Bytes

8.2.2.1 Characters and Words

A sequence model reads one token per time step. The two obvious choices of token are characters and words. Let us try both, characters first.

```
@d2l.add_to_class(TimeMachine)
def _tokenize(self, text):
    return list(text)

tokens = data._tokenize(text)
','.join(tokens[:30])
```

```
't,h,e, ,t,i,m,e, ,m,a,c,h,i,n,e, ,b,y, ,h, ,g, ,w,e,l,l,s, '
```

Splitting on whitespace instead yields word tokens. For each scheme we count the *distinct* token values (the vocabulary a model must handle) and the total number of tokens (the length of the sequence it must process).

```
words = text.split()
print(f'characters: {len(set(tokens)):>6} distinct, '
      f'{len(tokens):>7} tokens in the corpus')
print(f'words:      {len(set(words)):>6} distinct, '
      f'{len(words):>7} tokens in the corpus')
```

```
characters:     27 distinct,  173428 tokens in the corpus
words:        4579 distinct,   32775 tokens in the corpus
```

Character tokenization of our normalized text needs only 27 symbols and never meets an unknown letter. But it stretches the corpus to some 170,000 time steps, and every long-range relationship (say, between a subject and its verb) must be carried across dozens of steps. Word tokenization compresses the same text into a fifth as many tokens, each far more meaningful. Already on this small corpus, however, it produces thousands of distinct types, and the vocabulary keeps growing with the corpus: every name, typo, and inflection ("traveller", "travellers", "travelled") claims its own slot, most of them rarely used (Section 8.3). Worse, any word not seen during training is *out of vocabulary* (OOV) at test time. Figure 8.2 places the two schemes on a common axis. A larger vocabulary gives shorter sequences, and the question is where between the extremes to sit. Subword tokenizers let us choose that point freely, and most of this section goes into building one.

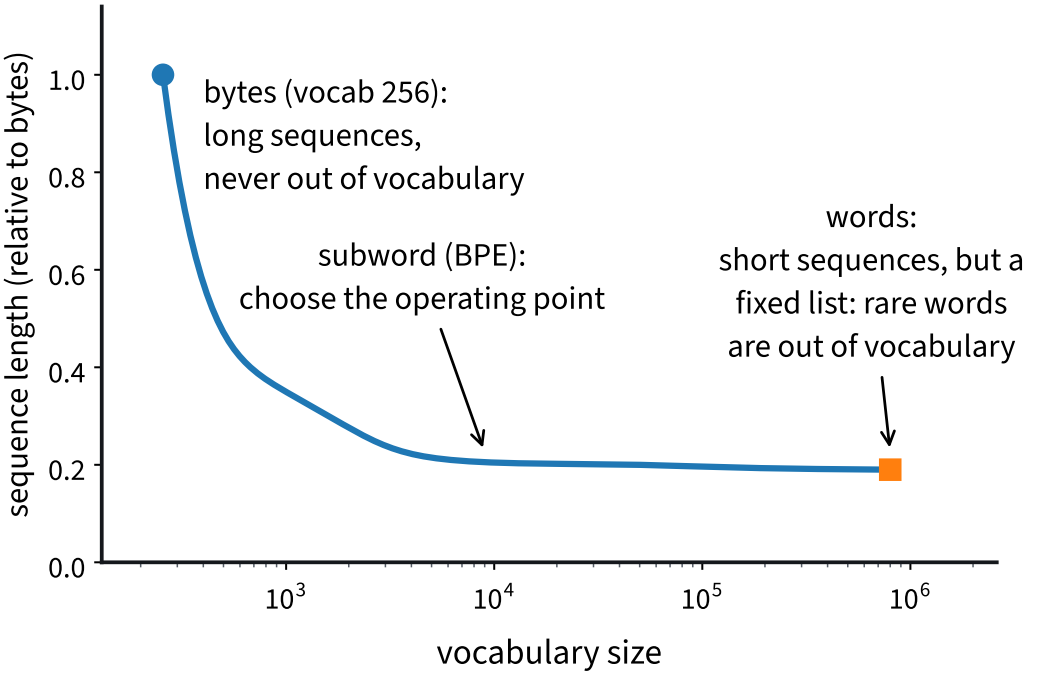


Figure 8.2: Tokenization granularities on one trade-off curve: a larger vocabulary encodes the same text in fewer tokens. Bytes never fall out of vocabulary; a fixed word list does. Byte pair encoding interpolates between the extremes.

8.2.2.2 From Tokens to Indices

First, though, we finish the classical pipeline, because a few later sections of the book still use it. A model consumes integer ids, not strings, so characters and words need a *vocabulary* object. It assigns each distinct token string an index, most frequent first, and maps unknown tokens to a reserved `<unk>` slot. Rare tokens (below `min_freq` occurrences) can be dropped to keep the table small; they then become `<unk>` as well.

```
class Vocab:
    """Vocabulary for text."""
```

```
    def __init__(self, tokens=(), min_freq=0, reserved_tokens=()):
        # Flatten a 2D list if needed
        if tokens and isinstance(tokens[0], list):
            tokens = [token for line in tokens for token in line]
        # Count token frequencies
        counter = collections.Counter(tokens)
        self.token_freqs = sorted(counter.items(), key=lambda x: x[1],
                                  reverse=True)
        # The list of unique tokens, ordered by descending frequency.
        # Reserve <unk> at index 0 so vocab[0] is the unknown token.
        self.idx_to_token = ['<unk>'] + list(reserved_tokens) + [
            token for token, freq in self.token_freqs
            if freq >= min_freq and token not in reserved_tokens]
        self.token_to_idx = {token: idx
                             for idx, token in enumerate(self.idx_to_token)}

    def __len__(self):
        return len(self.idx_to_token)

    def __getitem__(self, tokens):
        if not isinstance(tokens, (list, tuple)):
            return self.token_to_idx.get(tokens, self.unk)
        return [self.__getitem__(token) for token in tokens]

    def to_tokens(self, indices):
        if isinstance(indices, (list, tuple)) or getattr(indices, 'ndim', 0):
            return [self.idx_to_token[int(index)] for index in indices]
        return self.idx_to_token[int(indices)]

    @property
    def unk(self):  # Index for the unknown token
        return self.token_to_idx['<unk>']
```

```
vocab = Vocab(tokens)
indices = vocab[tokens[:10]]
print('indices:', indices)
print('tokens:', vocab.to_tokens(indices))
```

```
indices: [3, 9, 2, 1, 3, 5, 13, 2, 1, 13]
tokens: ['t', 'h', 'e', ' ', 't', 'i', 'm', 'e', ' ', 'm']
```

The `build` convenience method packages the whole classical pipeline, preprocess, tokenize, and index, returning a corpus of ids and its vocabulary.

```
@d2l.add_to_class(TimeMachine)
def build(self, raw_text, vocab=None):
    tokens = self._tokenize(self._preprocess(raw_text))
    if vocab is None: vocab = Vocab(tokens)
    corpus = [vocab[token] for token in tokens]
    return corpus, vocab
```

```
corpus, vocab = data.build(raw_text)
len(corpus), len(vocab)
```

```
(173428, 28)
```

8.2.2.3 Text as Bytes

To interpolate between characters and words we need a base alphabet that covers every string. Characters are a poor choice for it. Unicode defines around 150,000 of them, covering scripts, accents, and emoji, so a character-level vocabulary is not small at all once we leave English. Modern tokenizers go one level lower instead. Every string your computer stores is ultimately a sequence of *bytes* under some encoding, almost always UTF-8. A byte holds one of $2^8 = 256$ values, so a byte-level model needs a base vocabulary of only 256 symbols, and *no input can ever be out of vocabulary*: any string, in any language, with any typo, is some sequence of bytes.

UTF-8 is a variable-length code: it uses one byte for an ASCII character and up to four for rarer ones.

```
s = 'Hello, naïve café:     '
print(len(s), 'characters,', len(s.encode('utf-8')), 'bytes')
for ch in 'a é    '.split():
    print(repr(ch), '->', list(ch.encode('utf-8')))
n_bytes = len(raw_text.encode('utf-8'))
print(f'The novella: {n_bytes} bytes, {n_bytes / len(words):.1f} per word')
```

```
23 characters, 32 bytes
'a' -> [97]
'é' -> [195, 169]
' ' -> [228, 189, 160]
' ' -> [240, 159, 154, 128]
The novella: 178979 bytes, 5.5 per word
```

Unfortunately, the universal alphabet makes sequences long. Our novella runs to nearly 180,000 bytes, more than five for each of its words, and since "é" takes two bytes, the Chinese character three, and the emoji four, raw bytes stretch non-English text even further. Everything that follows keeps the universality of bytes while shortening the sequences.

8.2.3 Byte Pair Encoding

8.2.3.1 The Merge Rule

Byte pair encoding starts from the 256 byte symbols and grows the vocabulary greedily: count how often each *pair* of adjacent tokens occurs in the training corpus, merge the most frequent pair into a single new token, and repeat until the vocabulary reaches a target size (Sennrich et al. 2016). Each merge shortens the corpus by the number of occurrences it collapses, so BPE is a greedy compression algorithm, and its vocabulary is a learned dictionary of the corpus's most common fragments.

It is instructive to trace the algorithm by hand. Our toy corpus has five distinct words in eight occurrences:

```
hug pug pun bun hugs hug hug pug
```

The corpus is 32 bytes long. The most frequent adjacent pair is (u, g), which occurs six times (in "hug", "pug", and "hugs"), so the first merge creates the token ug. After replacing every occurrence, the most frequent pair is (h, ug), and so on:

step	most frequent pair	count	new token	corpus length
0	(start from bytes)			32
1	u, g	6	ug	26
2	h, ug	4 (tie)	hug	22
3	hug, ' '	3 (tie)	'hug '	19

Steps 2 and 3 are ties: (ug, ' ') also occurs four times at step 2, and (' ', p) and (' ', hug) also occur three times at step 3. We break a tie in favor of the pair that occurs first in the corpus.

Two properties can be read off the trace. First, merges compose: the token hug is built on top of ug, so the learned vocabulary forms a *merge tree* over the byte alphabet (Figure 8.3), and any new text can be retokenized by replaying the same merges. The word "hugs", for instance, becomes the two tokens hug and s, while the rare word "bun" stays as three raw bytes: input that no merge applies to simply falls back to bytes. Second, the third merge glued a word to the *space that follows it*. A trailing space is harmless in itself, but nothing stops a later merge from attaching the next word to it. On a real corpus this is exactly what happens, and we fix it below.

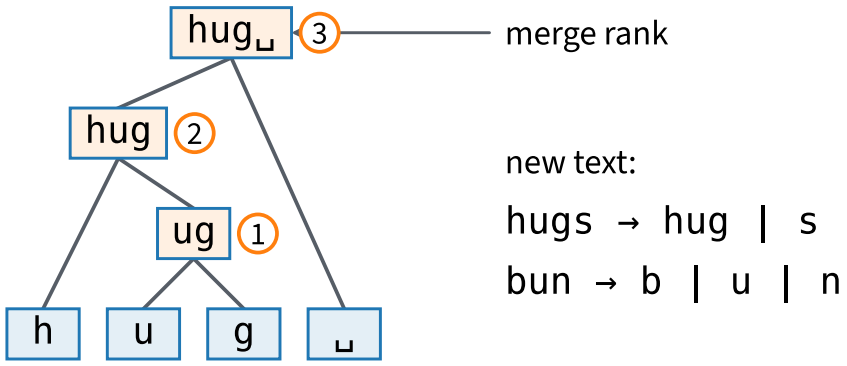


Figure 8.3: The first three merges learned on the toy corpus form a tree over the byte alphabet. New text reuses the tree: "hugs" becomes two tokens, while unseen fragments fall back to bytes.

8.2.3.2 Implementation

We implement the loop we traced in three parts. The class holds the state. Token ids 0 through 255 denote raw bytes, and each learned merge gets the next free id, so ids double as merge *ranks*: a smaller id means the merge was learned earlier. `merges` maps a pair of ids to the id of their merge, and `vocab` maps every id to its bytes. The `pattern` and `specials`

arguments and the table `byte_ids` are needed only later in this section and can be ignored for now. We import `regex`, an extension of Python's `re` whose Unicode character classes (such as `\p{L}` for "any letter") we need for pre-tokenization.

```
import regex

class BPETokenizer:
    """Byte-level BPE tokenizer trained by iterated most-frequent-pair
    merges.

    Token ids 0..255 are raw bytes; learned merges get ids
    256..vocab_size-1; special tokens sit above those and are never
    produced by BPE itself. `pattern` is an optional pre-tokenization
    regex: text is split into chunks and merges never cross chunk
    boundaries.
    """

    GPT2_PATTERN = (r"'s|'t|'re|'ve|'m|'ll|'d| ?\p{L}+| ?\p{N}+"
                    r"| ?[^\s\p{L}\p{N}]+|\s+(?!\S)|\s+")

    def __init__(self, vocab_size=1024, pattern=None,
                 specials=('<pad>', '<bos>', '<eos>')):
        self.vocab_size, self.pattern = vocab_size, pattern
        self.merges = {}                                # (id, id) -> new id
        self.vocab = {i: bytes([i]) for i in range(256)}  # id -> bytes
        self.byte_ids = list(range(256))                # byte value -> id
        self.specials = {s: vocab_size + i for i, s in enumerate(specials)}

    def __len__(self):
        return self.vocab_size + len(self.specials)

    @property
    def pad(self):
        return self.specials['<pad>']

    @property
    def bos(self):
        return self.specials['<bos>']

    @property
    def eos(self):
        return self.specials['<eos>']

    def _chunks(self, text):
        return [text] if self.pattern is None else regex.findall(
            self.pattern, text)
```

Training and encoding both rest on one operation: replace every occurrence of a pair of adjacent ids by the id of their merge. `_merge` looks at each position in turn. Where the pair starts, it writes the new id and skips two positions; elsewhere it copies one id. In `aaa` the pair (`a`, `a`) is therefore replaced once, at the left.

```
@d2l.add_to_class(BPETokenizer)
def _merge(self, ids, pair, new_id):
    """Replace every occurrence of pair in ids by new_id."""
    out, i = [], 0
    while i < len(ids):
        if i + 1 < len(ids) and (ids[i], ids[i + 1]) == pair:
            out.append(new_id)  # Merge the pair and skip past it
            i += 2
        else:
            out.append(ids[i])
            i += 1
    return out
```

Training repeats the step of the hand trace: count all adjacent pairs, merge the most frequent one everywhere, and record the merge. Ties go to the pair counted first, as in the trace, which is what `max` returns. The one refinement is that identical chunks of text are stored once, and their pair counts are weighted by how often the chunk occurs. For now the whole corpus is a single chunk and the weight is 1; once pre-tokenization splits text into words, a frequent word is counted once instead of thousands of times. Production trainers go further and update pair counts locally around each merge. We simply recount after every merge, which is slow and easy to read.

```
@d2l.add_to_class(BPETokenizer)
def train(self, text):
    """Learn vocab_size - 256 merges, most frequent pair first."""
    chunk_freq = collections.Counter(self._chunks(text))
    seqs = [list(chunk.encode('utf-8')) for chunk in chunk_freq]
    for new_id in range(256, self.vocab_size):
        pairs = collections.Counter()
        for seq, w in zip(seqs, chunk_freq.values()):
            for pair in zip(seq, seq[1:]):
                pairs[pair] += w  # A chunk that occurs w times counts w times
        if not pairs:
            break  # nothing left to merge
        pair = max(pairs, key=pairs.get)
        self.merges[pair] = new_id
        self.vocab[new_id] = self.vocab[pair[0]] + self.vocab[pair[1]]
        seqs = [self._merge(seq, pair, new_id) for seq in seqs]
```

Training on our toy corpus reproduces the hand trace:

```
toy_tok = BPETokenizer(vocab_size=256 + 3)
toy_tok.train('hug pug pun bun hugs hug hug pug')
for (a, b), new_id in toy_tok.merges.items():
    print(f'{toy_tok.vocab[a]} + {toy_tok.vocab[b]} -> '
          f'{toy_tok.vocab[new_id]}')
```

```
b'u' + b'g' -> b'ug'
b'h' + b'ug' -> b'hug'
b'hug' + b' ' -> b'hug '
```

8.2.3.3 Encoding and Decoding

Training produced a merge table. To *encode* new text we replay merges on its bytes: repeatedly find the adjacent pair with the lowest merge rank and merge it, until no adjacent pair is in the table. Encoding consults the ranks, not corpus statistics, so it is a pure function of the merge table: **the tokenizer is defined by its merge ranks**. Why lowest rank first? Training applied each merge to text in which all earlier merges had already been made, so replaying them in rank order retraces training (the exercises make this precise). Decoding is easier still. Every token id maps to a byte string, so we concatenate the strings and decode them as UTF-8. (A generation model can emit ids whose bytes do not align with character boundaries, so we ask Python to replace, rather than reject, malformed fragments.)

```
@d2l.add_to_class(BPETokenizer)
def _encode_chunk(self, text_bytes):
    ids = [self.byte_ids[b] for b in text_bytes]
    while len(ids) > 1:
        # The lowest-rank merge applicable anywhere in this chunk
        pair = min(zip(ids, ids[1:]),
                   key=lambda p: self.merges.get(p, float('inf')))
        if pair not in self.merges:
            break
        ids = self._merge(ids, pair, self.merges[pair])
    return ids

@d2l.add_to_class(BPETokenizer)
def encode(self, text, allow_special=False):
    if allow_special and self.specials:
        pat = '(' + '|'.join(regex.escape(s) for s in self.specials) + ')'
        ids = []
        for part in regex.split(pat, text):
            if part in self.specials:
                ids.append(self.specials[part])
            elif part:
                ids.extend(self.encode(part))
        return ids
    return [i for chunk in self._chunks(text)
            for i in self._encode_chunk(chunk.encode('utf-8'))]

@d2l.add_to_class(BPETokenizer)
def decode(self, ids):
    specials = {i: s.encode('utf-8') for s, i in self.specials.items()}
    data = b''.join(self.vocab[i] if i in self.vocab else specials[i]
                    for i in ids)
    return data.decode('utf-8', errors='replace')
```

On the toy tokenizer, encoding behaves as the hand trace predicted, and decoding inverts it.

```
print([toy_tok.vocab[i] for i in toy_tok.encode('hugs')])
print([toy_tok.vocab[i] for i in toy_tok.encode('bun')])
toy_tok.decode(toy_tok.encode('hug pug hugs'))
```

```
[b'hug', b's']
[b'b', b'u', b'n']
```

```
'hug pug hugs'
```

8.2.3.4 Training on The Time Machine

What does BPE learn from real prose? We train it on the novella. Token lists, and later generated text, run wider than a printed page, so we first write a utility that folds them, for use here and throughout the book.

```
import textwrap

def print_wrapped(*args, width=76):
    """Print like `print`, folding each line to the width of a page."""
    for line in ' '.join(str(a) for a in args).split('\n'):
        print(textwrap.fill(line, width, subsequent_indent='    '))
```

We now train a 1,024-token vocabulary on the raw, unnormalized novella. The first merges are letter pairs; by the end of training the tokenizer has learned whole words of Wells' prose. One of the tokens in the encoded sample, however, is the two-word composite "Time Traveller", spaces included. We return to it below.

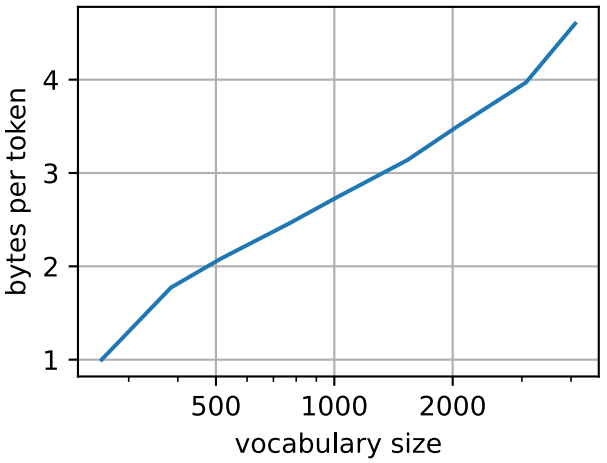
```
tok = BPETokenizer(vocab_size=1024)
tok.train(raw_text)
merged_ids = list(tok.merges.values())
print('first:', [tok.vocab[i] for i in merged_ids[:8]])
print('last: ', [tok.vocab[i] for i in merged_ids[-8:]])
ids = tok.encode('The Time Traveller smiled. Are you sure?')
print_wrapped(len(ids), 'tokens:', [tok.vocab[i] for i in ids])
```

```
first: [b'e ', b'th', b'd ', b'in', b't ', b's ', b'an', b'er']
last:  [b'appe', b'vo', b'ering ', b'too ', b'door', b'acro', b'clo', b'after ']
13 tokens: [b'The ', b'Time Traveller ', b's', b'mi', b'le', b'd', b'. A',
    b'r', b'e ', b'you ', b'su', b're', b'?']
```

How much shorter does a bigger vocabulary make the text? Because ids are assigned in rank order, a BPE tokenizer with a smaller vocabulary is a *prefix* of a larger one trained on the same corpus. For a vocabulary $\mathcal{V}$ of size $|\mathcal{V}|$, keeping only the merges with id below $|\mathcal{V}|$ recovers the tokenizer we would have gotten by stopping training at that size. A single training run therefore sweeps the vocabulary size from 256 to 4,096. We measure compression in bytes per token on the first 10,000 characters of the training text (exercise 7 measures held-out text).

```
big = BPETokenizer(vocab_size=4096)
big.train(raw_text)
excerpt = raw_text[:10000]
sizes = [256, 384, 512, 768, 1024, 1536, 2048, 3072, 4096]
ratios = []
for v in sizes:
    sub = BPETokenizer(vocab_size=v)  # the first v - 256 merges of big
```

```
    sub.merges = {p: i for p, i in big.merges.items() if i < v}
    sub.vocab = {i: b for i, b in big.vocab.items() if i < v}
    ratios.append(len(excerpt.encode('utf-8')) / len(sub.encode(excerpt)))
d2l.plot(sizes, [ratios], 'vocabulary size', 'bytes per token',
         xscale='log')
```



Each doubling of the vocabulary adds a roughly constant amount of compression: bytes per token grows about logarithmically in the vocabulary size. This is a consequence of the greedy rule. Early merges ("e", "th", "d") each remove thousands of occurrences, while the 4,000th merge removes a handful. It would be a mistake, however, to extend the line to production vocabularies of 50,000 to 200,000 tokens. The upper end of our curve is flattered: at 4,096 tokens the tokenizer has learned words and phrases of the very text it is measured on. On English prose at large, production vocabularies reach four to five bytes per token, about one token per word, as we measure below. Their extra entries go to rarer words, to code, and, in multilingual vocabularies such as `o200k_base`, to other languages.

8.2.3.5 Pre-Tokenization

The composite "Time Traveller" is not alone. Alongside word fragments, the vocabulary of the unconstrained tokenizer contains many tokens that straddle word boundaries:

```
crossers = [v.decode('utf-8', 'replace') for i, v in tok.vocab.items()
            if i >= 256 and b' ' in v.strip(b' ')]
print_wrapped(len(crossers), 'tokens cross a word boundary, e.g.',
              crossers[:8])
```

```
78 tokens cross a word boundary, e.g. [', and ', 'of the ', '. I ', '. Th',
    '. A', '. I', 'in the ', 'ed to ']
```

Merges like "of the" compress the training corpus, but they are brittle composites. They use up vocabulary slots on specific phrases, and they split the counts of the words they contain ("of" now appears in many tokens). Every production tokenizer therefore adds one step we have omitted so far, *pre-tokenization*. The text is first split into chunks with a regular expression, and BPE is trained and applied strictly *within* chunks, so no merge can cross a chunk

boundary. The pattern used by GPT-2 (Radford et al. 2019), kept as `BPETokenizer.GPT2_PATTERN`, reads as follows:

```
's|'t|'re|'ve|'m|'ll|'d        common English contractions, split off
| ?\p{L}+                      a run of letters, with its leading space
| ?\p{N}+                      a run of digits, with its leading space
| ?[^\s\p{L}\p{N}]+            a run of punctuation or other symbols
|\s+(?!\S)|\s+                 remaining whitespace
```

The pattern makes three decisions. A word keeps its *leading space* (" the" rather than "the"), so a word carries its own separation from the previous one. Letters, digits, and punctuation fall into separate chunks, so numbers and symbols never fuse with words. Contractions split off as units of their own.

```
regex.findall(BPETokenizer.GPT2_PATTERN,
              "The traveller's clock struck 12,345 times.")
```

```
['The',
 ' traveller',
 "'s",
 ' clock',
 ' struck',
 ' 12',
 ',',
 '345',
 ' times',
 '.']
```

Training with the pattern takes one more argument, and Figure 8.4 shows the resulting pipeline end to end. Compared with the unconstrained run, the boundary-crossing composites disappear, and their slots go to whole words with their leading space. The first merges now attach letters to a leading space (" t", " a") or join frequent letter pairs ("he", "in").

```
bpe = BPETokenizer(vocab_size=1024, pattern=BPETokenizer.GPT2_PATTERN)
bpe.train(raw_text)
crossers = [v for i, v in bpe.vocab.items()
            if i >= 256 and b' ' in v.strip(b' ')]
print('boundary-crossing tokens now:', len(crossers))
gained = sorted(set(bpe.vocab.values()) - set(tok.vocab.values()),
                key=lambda v: (-len(v), v))
print_wrapped('longest tokens gained instead:',
              [v.decode('utf-8', 'replace') for v in gained[:6]])
print('first merges:', [bpe.vocab[i] for i in list(bpe.merges.values())[:6]])
```

```
boundary-crossing tokens now: 0
longest tokens gained instead: [' Psychologist', 'sychologist', '
    Traveller', ' Morlocks', ' darkness', ' Machine']
first merges: [b' t', b'he', b' a', b'in', b' the', b' s']
```

8.2.4 Special Tokens

Training pipelines need a handful of tokens that mean something to the *infrastructure* rather than to the text. `<pad>` fills the unused tail of short sequences in a batch, `<bos>` marks the

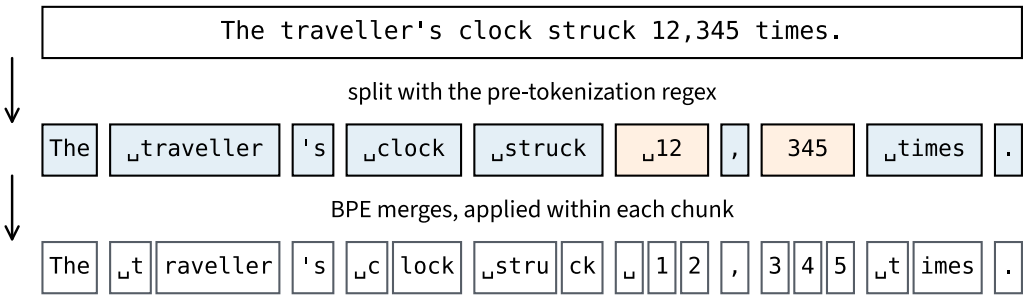


Figure 8.4: Pre-tokenization in action on a real sentence. The regex first cuts the text into chunks: words with their leading spaces, digit runs, and punctuation. BPE merges are then applied within each chunk, and can never cross a chunk boundary.

beginning of a sequence, and <eos> lets a model declare that generation is finished (the decoders of Section 8.7 accept it as a signal to stop). Our `BPETokenizer` assigns these ids *above* the BPE range. The placement is a safety property: encoding ordinary text must never produce a special token. If it could, any document that happened to contain the string "<eos>" could inject a control signal into the model's input. This is a form of *prompt injection*, in which text supplied by a user is read by the model as an instruction. Our `encode` therefore treats special strings in ordinary text as plain bytes unless the caller opts in with `allow_special=True`. `tiktoken` implements the same guard with its `allowed_special` argument and raises an error by default.

```
print('as plain text:  ', bpe.encode('<bos>the time machine'))
print('as control code:', bpe.encode('<bos>the time machine',
                                     allow_special=True))
print('pad/bos/eos ids:', bpe.pad, bpe.bos, bpe.eos, '| len:', len(bpe))
```

```
as plain text:   [60, 98, 111, 115, 62, 341, 513, 680]
as control code: [1025, 341, 513, 680]
pad/bos/eos ids: 1024 1025 1026 | len: 1027
```

8.2.5 Tokenizers in the Wild

8.2.5.1 The Rank Table of a Production Tokenizer

Most deployed language models, the GPT and Llama families among them, use BPE tokenizers of the kind we built, trained on far more text. GPT-2's tokenizer (Radford et al. 2019) is stored as a table that maps the bytes of each token to its rank, together with the pre-tokenization pattern and the special tokens. There is no neural network inside it. Other tokenizer families also store normalization rules, or a probabilistic model that chooses among segmentations. OpenAI's `tiktoken` library ships the GPT-2 table. We print the entries around the end of the byte range; the leading underscore of `_mergeable_ranks` marks a private attribute that may change between library versions.

```
enc = tiktoken.get_encoding('gpt2')
print(list(enc._mergeable_ranks.items())[254:260])
print(enc._special_tokens)
```

```
[(b'\xa0', 254), (b'\xad', 255), (b' t', 256), (b' a', 257), (b'he', 258), (b'in', 259)]
{'<|endoftext|>': 50256}
```

The first 256 ranks are the single bytes, in a permuted order (GPT-2 lists printable characters first), and from rank 256 on come the learned merges in the order they were learned: the structure of our `BPETokenizer`. GPT-2's first four merges, learned from 40 gigabytes of web text, are " t", " a", "he", and "in", the same four pairs that `bpe` learned first from one novella. The most frequent fragments of English are the same in both corpora.

8.2.5.2 Verifying Our Implementation

Since the tokenizer is a rank table, we can load GPT-2's table into our own encoder. Ranks become token ids, and the 256 single-byte entries give the byte permutation. The table stores the bytes of each token, not the pair that was merged to form it, so we recover the two parents of a longer token by re-running BPE on its bytes with all lower-ranked merges: they are whatever the bytes merge into just before the token's own rank applies. This loop can be skipped; what matters is the check that follows.

```
@d2l.add_to_class(BPETokenizer)
@classmethod
def from_tiktoken(cls, name):
    """Load a published bytes->rank table (e.g. 'gpt2') into our encoder."""
    import tiktoken  # lazy: d2l itself does not require tiktoken
    enc = tiktoken.get_encoding(name)
    ranks = enc._mergeable_ranks
    tok = cls(vocab_size=len(ranks), pattern=enc._pat_str, specials=())
    tok.specials = dict(enc._special_tokens)
    tok.vocab = {rank: b for b, rank in ranks.items()}
    tok.byte_ids = [ranks[bytes([b])] for b in range(256)]
    for token, rank in ranks.items():
        if len(token) > 1:  # recover which pair merged into this token
            parts = [bytes([b]) for b in token]
            while len(parts) > 2:
                a, b = min(zip(parts, parts[1:]),
                           key=lambda p: ranks.get(p[0] + p[1],
                                                   float('inf')))
                i = list(zip(parts, parts[1:])).index((a, b))
                parts[i:i + 2] = [a + b]
            tok.merges[ranks[parts[0]], ranks[parts[1]]] = rank
    return tok
```

On a paragraph that mixes prose, numbers, accents, and an emoji, our encoder reproduces the output of `tiktoken` token for token. The `assert` below checks this every time the book is built.

```
ours = BPETokenizer.from_tiktoken('gpt2')
para = ('The Time Traveller (for so it will be convenient to speak of '
        'him) was expounding a recondite matter to us; his pale grey '
        'eyes shone and twinkled. Prices rose 3.5% in 1895, and naïve '
        'café patrons paid £12,345 more!  ')
assert ours.encode(para) == enc.encode(para)
print(len(ours.encode(para)), 'tokens, identical to tiktoken')
print([ours.vocab[i].decode('utf-8', 'replace')
       for i in ours.encode(para)[:8]])
```

```
60 tokens, identical to tiktoken
['The', ' Time', ' Trave', 'ller', ' (', 'for', ' so', ' it']
```

How do production vocabularies fare on our novella? We compare our 1,024-token tokenizer with GPT-2's 50,257 tokens and with `o200k_base`, the vocabulary of about 200,000 tokens used by GPT-4o.

```
rows = [('ours (1k, in-domain)', len(bpe.encode(raw_text)))]
for name in ('gpt2', 'o200k_base'):
    e = tiktoken.get_encoding(name)
    rows.append((f'{name} ({len(e._mergeable_ranks) // 1000}k)',
                 len(e.encode(raw_text))))
for name, n in rows:
    print(f'{name:>22}: {n:>6} tokens, {n_bytes / n:.2f} bytes/token')
```

```
  ours (1k, in-domain):  65741 tokens, 2.72 bytes/token
           gpt2 (50k):  44073 tokens, 4.06 bytes/token
    o200k_base (199k):  42624 tokens, 4.20 bytes/token
```

Although it is 50 to 200 times smaller, our tokenizer reaches about two thirds of their compression, because it was trained on this text. A tokenizer is a compression model of its training corpus, and like any model it does best on data like its training data.

8.2.5.3 Fertility, Digits, and Glitch Tokens

The number of tokens a tokenizer uses per word of a language is its *fertility*. GPT-2's tokenizer was trained on mostly English text, and the same sentence in three languages takes very different numbers of tokens:

```
sentences = {
    'English': 'The time traveller returned after many years.',
    'German': 'Der Zeitreisende kehrte nach vielen Jahren zurück.',
    'Greek': '                                                  .'}
for lang, s in sentences.items():
    gpt2, o200k = [len(tiktoken.get_encoding(e).encode(s)) / len(s.split())
                   for e in ('gpt2', 'o200k_base')]
    print(f'{lang:>8}: {gpt2:.2f} (gpt2), {o200k:.2f} (o200k_base) '
          'tokens per word')
```

```
English: 1.14 (gpt2), 1.14 (o200k_base) tokens per word
 German: 2.86 (gpt2), 1.57 (o200k_base) tokens per word
  Greek: 7.33 (gpt2), 1.89 (o200k_base) tokens per word
```

Under GPT-2's tokenizer a Greek word takes about seven tokens (Greek letters take two UTF-8 bytes each, and few Greek merges were learned), an English word about one. A 1,024-token context window therefore holds about 140 Greek words against about 900 English ones. The multilingual `o200k_base` brings Greek down to about two tokens per word.

Digits show how small tokenization choices change what a model must learn. GPT-2 splits digit strings wherever BPE found frequent fragments; newer tokenizers cap digit runs at three during pre-tokenization, which makes the split predictable:

```
for name in ('gpt2', 'o200k_base'):
    e = tiktoken.get_encoding(name)
    print(f'{name}: 12345678 ->',
          [e.decode([t]) for t in e.encode('12345678')])
```

```
gpt2: 12345678 -> ['123', '45', '678']
o200k_base: 12345678 -> ['123', '456', '78']
```

Neither split aligns tokens with the columns of written addition, which works right to left in ones, tens, and hundreds. Under the cap, "12345678" splits left to right into "123", "456", "78". A model must learn arithmetic over such misaligned chunks. This is one reason why language models make arithmetic errors, and why writing numbers with separators ("12,345,678") can help.

Finally, a token is only as useful as the model's training on it. Early GPT-2 and GPT-3 vocabularies contain tokens such as " SolidGoldMagikarp", a Reddit username that was frequent in the corpus the *tokenizer* was trained on but rare in the corpus the *language model* was trained on. Such a *glitch token* can appear in a prompt, yet its embedding was barely trained. Asked to repeat it, models produced evasions and unrelated words.

```
enc.encode(' SolidGoldMagikarp')  # one token for an 18-character string
```

```
[43453]
```

8.2.6 Summary

A tokenizer turns text into the integer sequence a model reads. Its vocabulary size trades the length of that sequence against the number of entries the model must learn, and a fixed word list also fails on unseen words. Byte-level BPE starts from the 256 byte values, so no input is out of vocabulary, and greedily merges the most frequent adjacent pair until the vocabulary is full. Encoding replays the merges in rank order, so the merge table is the tokenizer. A pre-tokenization pattern keeps merges inside words, and special tokens sit outside the BPE range so that ordinary text cannot produce them. GPT-2's tokenizer is the same construction trained on more text, and our encoder reproduces it token for token. Its fertility across languages, its digit chunks, and its glitch tokens all follow from how it was trained. The next section models the token sequences that such a tokenizer produces.

8.2.7 Exercises

1. **Encoding by rank.** Suppose a `BPETokenizer` with `vocab_size=258` has learned exactly two merges: (`b`, `c`) with id 256, and then (`a`, `b`) with id 257. The *rank rule* of `_encode_chunk`

always applies the lowest-ranked applicable merge. The *leftmost rule* always merges the leftmost adjacent pair that appears in the merge table.

1. Segment the string "abc" under each of the two rules.
2. Give the segmentation that every occurrence of "abc" in the training corpus had at the end of training, and say which rule agrees with it.

2. code **Cross-corpus compression.** Train a `BPETokenizer` with a 1,024-token vocabulary and `BPETokenizer.GPT2_PATTERN` on a different text of your choice, such as another public-domain book, your own notes, or source code.

   1. Compute the fraction of its first 100 merged tokens, and of its last 100, that are also tokens of `bpe`. Explain the difference between the two fractions.
   2. Measure bytes per token of your tokenizer on *The Time Machine* and of `bpe` on your corpus, and compare both with the in-domain figures.

3. code **Special-token round trip.** Section 8.2.4 places special tokens above the BPE id range so that ordinary text can never produce them. Construct a `BPETokenizer` whose `specials` add a document separator `<sep>` to the three defaults, and train it on `raw_text`. Let `x` be two short documents joined by `<sep>`.

   1. Encode `x` once with and once without `allow_special=True`, and check that both id sequences decode to `x`.
   2. Re-encode the decoded text once with `allow_special=True` and once without it. Which of the two id sequences from the previous sub-problem does each re-encoding reproduce? State what this implies for a pipeline that decodes a model's output and later feeds the text back in as input.
   3. `tiktoken` raises an error when a special string appears in text that is not allowed to contain it, whereas `encode` treats the string as plain bytes. Give one situation in which each policy is preferable.

   Adapted from Andrej Karpathy, minbpe, Step 4.

4. code **Digit chunking.** Tokenize the addition prompts "1234+5678=", "1234+567=" and "567+678=" and their sums "6912", "1801" and "1245" with the `gpt2` and `o200k_base` encodings of `tiktoken`, writing out the token boundaries of each number.

   1. Under the three-digit cap of `o200k_base`, label each digit token with the place values (ones, tens, hundreds, thousands) it covers. For each prompt, state whether the tokens at matching positions of the two summands and of the sum cover the same place values. Explain what goes wrong for a model that adds token by token when they do not.
   2. Propose a pre-tokenization rule that aligns tokens with place value, and state one drawback of your rule.

5. **Glitch tokens.** Section 8.2.5.3 attributes a glitch token to a string that was frequent in the tokenizer's training corpus but rare in the language model's.

   1. Why does the merge rule of `BPETokenizer.train` turn such a string into a single token rather than several fragments, and what role does pre-tokenization play in fixing the token's extent?
   2. Propose a test that flags glitch tokens given the trained embedding table and the model's training corpus, and state what each of the two inputs contributes.

6. **Rank replay.** ▯ The *rank rule* of `_encode_chunk` always applies the lowest-ranked applicable merge. Argue that, for any merge table learned by `BPETokenizer.train`, the rank rule reproduces the end-of-training segmentation of every chunk of the training corpus.

7. code **Pre-tokenization ablation.** ▯ Split `raw_text` into its first 90% and its last 10%. On the first part, train two 4,096-token tokenizers that differ only in the pattern: one without a pattern, as `big` was trained, and one with `BPETokenizer.GPT2_PATTERN`.

    1. List the 20 longest tokens learned by each tokenizer, and mark those that span more than one word.
    2. Measure which of the two compresses the first part better, and explain the outcome in terms of the merge rule.
    3. Measure bytes per token of both tokenizers on the last 10%. Which tokenizer loses the larger fraction of its compression on this held-out text, and which compresses it better? Relate both answers to the tokens you listed.

8.3 Language Models

A phone keyboard that proposes the next word, and a speech recognizer that must decide whether it heard “to recognize speech” or “to wreck a nice beach”, need the same thing: a number that says how plausible a piece of text is. The two phrases sound almost alike, so the acoustics cannot settle the question. What settles it is knowing that the first is a common English phrase and the second is not. A *language model* supplies this number: given a sequence of tokens $x_1, \dots, x_T$ from a vocabulary $\mathcal{V}$, such as the BPE tokens of Section 8.2, it assigns the sequence a probability $P(x_1, \dots, x_T)$.

Assigning a probability to every sequence looks hopeless at first. With our vocabulary of 1,024 tokens there are $1024^{10} \approx 10^{30}$ sequences of only ten tokens. The chain rule of probability, Equation 8.2 in Section 8.1, turns the problem into one we know. It writes the joint probability, with no approximation, as a product of next-token predictions, each conditioned on the prefix $x_{<t} = x_1, \dots, x_{t-1}$:

$$P(x_1, \dots, x_T) = \prod_{t=1}^{T} P(x_t \mid x_{<t}). \tag{8.5}$$

A language model is therefore a next-token predictor. The same conditionals score a sequence (multiply them) and generate one (draw x_1, then x_2 given x_1, and so on). Surprisingly many tasks can be phrased as a continuation as well. Continuing “The novella The Time Machine was written by” answers a question, and this is how large language models are used (Section 11.2).

Let us start with the simplest estimate of the conditionals, counting, and find out where it fails and why. We then define a number that says how good a language model is, use it to judge two repairs of the counts, and free it from the choice of tokenizer. Last, we prepare the training data for the neural language models of the following sections.

```
import collections
import random
import re
```

8.3.1 n-gram Language Models

8.3.1.1 Markov Models and Counting

The conditionals in Equation 8.5 depend on the whole prefix, so the number of distinct contexts grows without bound as t grows. The Markov condition of Section 8.1.3.1 cuts the context to the most recent $n-1$ tokens:

$$P(x_t \mid x_{<t}) \approx P(x_t \mid x_{t-n+1}, \dots, x_{t-1}).$$

The assumption is crude, since a sentence does not forget its subject after two words, but it leaves a finite number of contexts to estimate. A model with no context ($n = 1$) is called a *unigram* model, one that conditions on the previous token ($n = 2$) a *bigram* model, and one that conditions on the previous two ($n = 3$) a *trigram* model. The conditional probabilities are the model's parameters, and maximum likelihood estimates each of them as a ratio of counts. Writing $N(\cdot)$ for the number of occurrences of a sequence of tokens in the training corpus,

$$\hat{P}(x_t \mid x_{t-n+1}, \dots, x_{t-1}) = \frac{N(x_{t-n+1}, \dots, x_t)}{N(x_{t-n+1}, \dots, x_{t-1})}. \tag{8.6}$$

Thus the estimate is the fraction of the occurrences of the context that were followed by x_t. An n-gram language model is nothing but a table of these ratios.

Let us build such tables for *The Time Machine*. The neural language models of this chapter read BPE tokens, but here we split the text into words. The reason is simply that we want to read what the models generate, and words read more easily than subword fragments. We repeat the download and normalization of Section 8.2 (letters only, lowercase), split on whitespace, and hold out the final 10% of the text to judge the models on text they did not see. The vocabulary consists of the words of the training text; a held-out word that training never saw becomes the unknown token `<unk>`.

```
fname = d2l.download(d2l.DATA_URL + 'timemachine.txt',
                     sha1_hash='090b5e7e70c295757f55df93cb0a180b9691891a')
raw_text = open(fname).read()
text = re.sub('[^A-Za-z]+', ' ', raw_text).lower()
split = text.rindex(' ', 0, int(0.9 * len(text)))
words_train = text[:split].split()
known = set(words_train)  # A held-out word never seen in training is <unk>
words_val = [w if w in known else '<unk>' for w in text[split:].split()]
len(words_train), len(words_val)
```

```
(29402, 3373)
```

The class below stores, for every context of $n-1$ tokens, a counter of the tokens that followed it in training. That is all Equation 8.6 needs. (The constructor also stores a constant `alpha`, which plays no role until we smooth the counts.) `sample` extends a prefix by drawing each next token in proportion to its count, looking only at the last $n-1$ tokens of the text so far.

```
class NGramLM:
    """An n-gram language model estimated from counts."""
    def __init__(self, tokens, n, vocab_size, alpha=0.01):
        self.n, self.vocab_size, self.alpha = n, vocab_size, alpha
        cont = collections.defaultdict(collections.Counter)
        for gram in zip(*(tokens[i:] for i in range(n))):
            cont[gram[:-1]][gram[-1]] += 1
        self.cont = dict(cont)
        self.total = {ctx: sum(c.values()) for ctx, c in self.cont.items()}

    def sample(self, prefix, num_tokens, rng):
        """Extend prefix by sampling from the unsmoothed count model."""
        out = list(prefix)
        for _ in range(num_tokens):
            ctx = tuple(out[len(out) - self.n + 1:])
            counts = self.cont[ctx]
            out.append(rng.choices(list(counts), counts.values())[0])
        return out

vocab_size = len(known) + 1  # The training words and <unk>
models = {n: NGramLM(words_train, n, vocab_size) for n in (1, 2, 3)}
```

What do the counts look like? We take the trigram table and look up the context “the time”, which occurs often in a novel about time travel.

```
trigram = models[3]
total = trigram.total[('the', 'time')]
for word, count in trigram.cont[('the', 'time')].most_common(3):
    print(f'P({word} | the time) = {count}/{total} = {count / total:.2f}')
single = sum(len(c) == 1 for c in trigram.cont.values()) / len(trigram.cont)
print(f'word pairs followed by a single word in training: {single:.0%}')
```

```
P(traveller | the time) = 47/87 = 0.54
P(machine | the time) = 27/87 = 0.31
P(i | the time) = 6/87 = 0.07
word pairs followed by a single word in training: 84%
```

The pair “the time” occurs 87 times in the training text and is followed by “traveller” in about half of them and by “machine” in about a third, so the model has learned that the novel’s protagonist is the Time Traveller. Common contexts like this one are the exception, though. For more than four in five word pairs the table lists a single successor: the model has seen only one way to continue them.

8.3.1.2 Sampling from n-gram Models

Generation follows Equation 8.5: draw the next token from the model’s conditional given the text so far, append it, and repeat. It is tempting to judge a model by the text it writes. As a precaution, we also measure the longest run of consecutive sampled words that occurs verbatim in the training text.

```
def longest_copy(words, train_words):
    """Length of the longest run of words that also occurs in train_words."""
    corpus, best = ' ' + ' '.join(train_words) + ' ', 0
    for i in range(len(words)):
        while (i + best < len(words) and
               ' ' + ' '.join(words[i:i + best + 1]) + ' ' in corpus):
            best += 1
    return best
```

We hand each model the prefix "the time" and let it generate 25 words. The random number generator is not seeded, so every run draws new samples.

```
rng = random.Random()
for n, model in models.items():
    sample = model.sample(['the', 'time'], 25, rng)
    copied = longest_copy(sample, words_train)
    d2l.print_wrapped(f'{n}-gram (longest copied run: {copied} words): '
                      + ' '.join(sample))
```

```
1-gram (longest copied run: 3 words): the time a and and more to bar
    buildings seen that refrained decay fire inhuman a the my there
    curiously me at the thrust have air me
2-gram (longest copied run: 5 words): the time and the palace proved on the
    tenderness for the little people soon stop examining me and one very
    high yesterday night by the silent man
3-gram (longest copied run: 8 words): the time traveller s words we should
    call the three new guests but the day came on and explore but the
    twisted crystalline bars lay unfinished upon
```

The unigram model emits words with the right frequencies but no relation to one another. The bigram model produces plausible pairs that drift without direction, since each word depends only on its predecessor. The trigram text reads like Wells.

Unfortunately, on a corpus of 30,000 words it reads like Wells because much of it *is* Wells. The longest copied run measures how much. It grows with the order: two or three words for the unigram, a few more for the bigram, and typically seven to a dozen for the trigram. We have seen why. A word pair with a single successor leaves the trigram model no choice but to continue as Wells did, so it replays clauses of the novella verbatim. It has memorized its training data rather than generalized from it. Trained on billions of words, the same tables do capture much of the local syntax of English, which is why n-gram models served speech recognition and translation systems for decades.

8.3.1.3 Count Sparsity

Memorization is a symptom of a quantitative problem. The number of *possible* n-grams is $|\mathcal{V}|^n$, which grows exponentially in n, while the number of n-grams a corpus contains grows only linearly with its length. We measure the resulting sparsity directly: for each order, which fraction of the held-out n-grams has a training count of zero?

```
for n in (1, 2, 3):
    seen = set(zip(*(words_train[i:] for i in range(n))))
```

```
    grams = list(zip(*(words_val[i:] for i in range(n))))
    rate = sum(gram not in seen for gram in grams) / len(grams)
    print(f'unseen held-out {n}-grams: {rate:.1%} '
          f'({len(seen)} distinct in training)')
```

```
unseen held-out 1-grams: 8.4% (4315 distinct in training)
unseen held-out 2-grams: 51.5% (18535 distinct in training)
unseen held-out 3-grams: 87.0% (26982 distinct in training)
```

About one held-out word in twelve never occurs in training. For word *pairs* it is one in two, and for word triples nearly nine in ten. The maximum likelihood estimate Equation 8.6 gives each of them probability zero, and a single zero in the product Equation 8.5 declares impossible a text that we hold in our hands.

8.3.1.4 Word Frequency and Zipf's Law

Why do the count tables go sparse so fast? The answer lies in how word frequencies are distributed. We use the `Vocab` class of Section 8.2, which orders tokens by frequency, to list the most common words of *The Time Machine*.

```
words = text.split()
vocab = d2l.Vocab(words)
vocab.token_freqs[:10]
```

```
[('the', 2261),
 ('i', 1267),
 ('and', 1245),
 ('of', 1155),
 ('a', 816),
 ('to', 695),
 ('was', 552),
 ('in', 541),
 ('that', 443),
 ('my', 440)]
```

The ten most frequent words say nothing about time travel. They are articles, pronouns, and prepositions, so-called *stop words*, and any Victorian novel would yield much the same list. More interesting is how fast the frequencies fall: the tenth most common word is less than a fifth as frequent as the most common one. On log-log axes, frequency against rank is close to a straight line past the first few words. A straight line on log-log axes is a power law: the frequency n_i of the i^{th} most frequent word obeys

$$n_i \propto \frac{1}{i^s}, \quad \text{equivalently} \quad \log n_i = -s \log i + c, \tag{8.7}$$

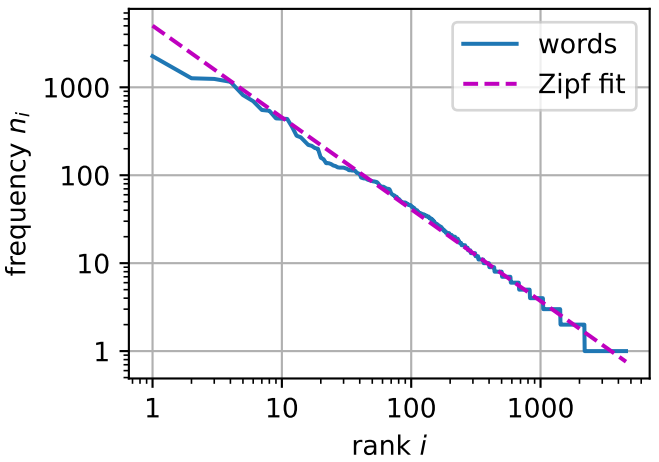
for an exponent s and a constant c. This is *Zipf's law*. We estimate s by a least-squares fit of the line over ranks 10 to 1,000 and draw the fitted law next to the data.

```
def zipf_fit(freqs, lo=10, hi=1000):
    """Fit log n_i = -s log i + c over ranks lo..hi; return (s, c)."""
    ranks = np.arange(lo, hi + 1)
```

```
    slope, c = np.polyfit(np.log(ranks), np.log(freqs[lo - 1:hi]), 1)
    return -slope, c
```

```
freqs = [freq for token, freq in vocab.token_freqs]
s, c = zipf_fit(freqs)
print(f'Zipf exponent s = {s:.2f}')
ranks = np.arange(1, len(freqs) + 1)
d2l.plot(ranks, [freqs, np.exp(c) * ranks ** -s], xlabel='rank $i$',
         ylabel='frequency $n_i$', xscale='log', yscale='log',
         legend=['words', 'Zipf fit'])
```

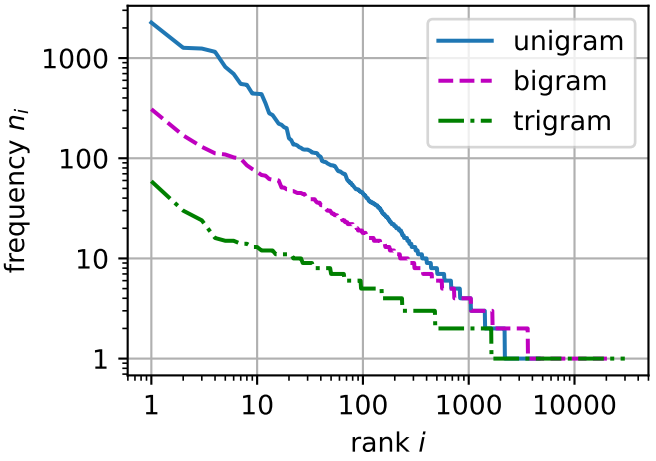
```
Zipf exponent s = 1.04
```



The exponent is close to 1, the value Zipf reported for word frequencies in many languages. A power law has a heavy tail: each rare word is rare, but there are so many of them that together they make up a large part of any text. Does the same hold for word pairs and triples? We count the n-grams of each order, fit the exponent, measure which fraction of the distinct n-grams occurs only once, and draw the three curves on common axes.

```
counts = [sorted(collections.Counter(zip(*(words[i:] for i in range(n))))
                 .values(), reverse=True) for n in (1, 2, 3)]
for n, f in zip((1, 2, 3), counts):
    print(f'{n}-grams: s = {zipf_fit(f)[0]:.2f}, '
          f'{np.mean(np.array(f) == 1):.0%} of distinct {n}-grams occur once')
d2l.plot([np.arange(1, len(f) + 1) for f in counts], counts,
         xlabel='rank $i$', ylabel='frequency $n_i$', xscale='log',
         yscale='log', legend=['unigram', 'bigram', 'trigram'])
```

```
1-grams: s = 1.04, 52% of distinct 1-grams occur once
2-grams: s = 0.69, 82% of distinct 2-grams occur once
3-grams: s = 0.47, 95% of distinct 3-grams occur once
```



Pairs and triples follow power laws too, with smaller exponents, about 0.7 and 0.5. More important for count models, the curves shift toward count one as n grows. Half of the distinct words occur once, but more than nine in ten distinct trigrams do, and each of their estimates rests on a single observation. This is the heavy tail behind the unseen trigrams of the held-out text. A model that estimates the probability of each context separately needs exponentially more data as the context grows.

8.3.2 Perplexity and Smoothing

8.3.2.1 Perplexity

The count tables are mostly empty, and the samples hid it from us. Before we repair the tables we need a number that says how good a language model is, so that we can tell whether a repair has worked. The probability that a model assigns to held-out text is the natural candidate, but it shrinks with the length of the text: any model gives Tolstoy's *War and Peace* a far smaller probability than Saint-Exupéry's *The Little Prince*, whatever its quality. We need an average per token.

Information theory provides it. In Section 3.1.2.3, the cross-entropy loss of a classifier was the *surprisal* of the true label, the negative logarithm of the probability the classifier assigned to it. A language model is a classifier over the next token, so we score it by its surprisal averaged over all T tokens of the text,

$$\frac{1}{T}\sum_{t=1}^{T} -\log P(x_t \mid x_{<t}), \tag{8.8}$$

where P is given by the language model and x_t is the token observed at step t. This average no longer depends on the length of the text. Natural language processing reports its exponential, called *perplexity* (Jelinek et al. 1977):

$$\mathrm{PPL} = \exp\left(-\frac{1}{T}\sum_{t=1}^{T}\log P(x_t \mid x_{<t})\right).$$

Thus perplexity is the geometric mean, over the tokens of the text, of the inverse probability that the model assigned to each of them. A model that gives every observed token probability $1/4$ has perplexity 4: it is as uncertain as a fair choice among four tokens. Perplexity is therefore read as the *effective branching factor*, the number of equally likely choices the model faces per token. Three reference points fix the scale:

- A model that always assigns the observed token probability 1 has perplexity 1, the best possible value.
- A model that assigns some observed token probability 0 has infinite perplexity.
- A model that predicts the uniform distribution over the vocabulary has perplexity $|\mathcal{V}|$. Any useful model must do better.

The second case is ours. The maximum likelihood estimate gives every unseen n-gram probability zero, and a single unseen trigram in the held-out text makes the perplexity of the trigram model infinite. Before we can measure anything, the zeros must go: the counts must be *smoothed*.

8.3.2.2 Laplace Smoothing

The classical remedy is *Laplace smoothing*, also called add-α smoothing. It adds a constant $\alpha > 0$ to every count and $\alpha|\mathcal{V}|$ to the denominator, so that the probabilities still sum to one:

$$\hat{P}(x_t \mid x_{t-n+1}, \ldots, x_{t-1}) = \frac{N(x_{t-n+1}, \ldots, x_t) + \alpha}{N(x_{t-n+1}, \ldots, x_{t-1}) + \alpha|\mathcal{V}|}. \tag{8.9}$$

An unseen n-gram now gets a small positive probability, and an unseen context gets the uniform distribution $1/|\mathcal{V}|$. The idea is two centuries older than language modeling: Laplace introduced it, with $\alpha = 1$, to estimate the probability that the sun will rise tomorrow given that it has risen on every day so far.

We add Laplace smoothing to `NGramLM` as a method `prob`, which uses the `alpha` that the constructor stored, and evaluate the three word-level models on the held-out text.

```
@d2l.add_to_class(NGramLM)
def prob(self, context, token):
    """Laplace-smoothed P(token | last n-1 tokens of context)."""
    ctx = tuple(context[len(context) - self.n + 1:])
    num = self.cont.get(ctx, {}).get(token, 0) + self.alpha
    return num / (self.total.get(ctx, 0) + self.alpha * self.vocab_size)
```

```
def eval_nll(model, tokens):
    """Total negative log-likelihood of tokens, in nats."""
    return -sum(math.log(model.prob(tokens[max(0, t - model.n + 1):t],
                                    tokens[t]))
                for t in range(len(tokens)))

for n, model in models.items():
    ppl = math.exp(eval_nll(model, words_val) / len(words_val))
    print(f'{n}-gram held-out perplexity, Laplace smoothing: {ppl:7.1f}')
```

```
1-gram held-out perplexity, Laplace smoothing:   840.6
2-gram held-out perplexity, Laplace smoothing:   753.6
3-gram held-out perplexity, Laplace smoothing:  2481.4
```

With Laplace smoothing, the trigram model, which wrote by far the most convincing text, has by far the *worst* held-out perplexity. At the shared smoothing constant, the bigram barely improves on the unigram. Both results come from count sparsity. When 87% of the held-out trigrams are unseen, most of the trigram model's predictions are smoothed, nearly uniform guesses. No choice of α rescues it: the best value in the sweep of exercise 3 still leaves its perplexity above 2,400. The fluent samples were a poor guide.

Laplace smoothing is a crude remedy, and the numbers point to its two defects. It treats every unseen successor alike. And it ignores what the shorter contexts know: a trigram never seen in training gets the same probability whether its last two words form a common pair or not.

8.3.2.3 Kneser–Ney Smoothing

Kneser–Ney smoothing (Kneser and Ney 1995), in the interpolated form of Chen and Goodman (1999), repairs both defects, and it was the strongest count-based language model for two decades. Write u for a context of $n-1$ tokens, w for the token that follows it, and u' for u without its first token. The method rests on three ideas.

The first is *absolute discounting*. An n-gram that occurred once in training is, on average, seen less than once in new text of the same size: among all the n-grams that could have occurred once, those that did were partly lucky. Subtracting a fixed discount $0 < D < 1$ from every observed count corrects this, most strongly for the small counts where the error is largest. The probability mass freed in context u is $D\,S(u)/N(u)$, where $S(u)$ is the number of distinct tokens that followed u in training.

The second is *interpolation*: the freed mass goes to the estimate of the next lower order, which is defined in the same way, down to the uniform distribution $1/|\mathcal{V}|$:

$$P_{\text{KN}}(w \mid u) = \frac{\max(N(u,w) - D, 0)}{N(u)} + \frac{D\,S(u)}{N(u)}\,P_{\text{KN}}(w \mid u'). \tag{8.10}$$

Thus a context seen often relies mostly on its own counts. A context followed by many different tokens hands more mass to the shorter context, since so many distinct successors suggest that new ones are likely as well. A context never seen in training passes all of the mass down: $P_{\text{KN}}(w \mid u) = P_{\text{KN}}(w \mid u')$.

The third idea concerns what the lower orders count. The lower-order estimate matters mainly for successors that the longer context has not been seen with. It should therefore measure how readily a token follows a *new* context, not how often it occurs. In the training text, "traveller" occurs 49 times but after only two different words ("time" 48 times, "real" once), while "said" occurs 76 times after 48 different words. A raw count makes "traveller" a likely successor of any unseen context; the number of distinct predecessors says correctly that it is not. (The standard example is "Francisco", frequent but almost always preceded by "San".) Kneser–Ney therefore replaces $N(u,w)$ in every order below the highest by the *continuation count*, the number of distinct tokens that precede (u,w) in training, and $N(u)$ and $S(u)$ by the total and the number of nonzero continuation counts in context u.

The discount need not be tuned. Ney et al. (1994) derived the estimate $D = c_1/(c_1 + 2c_2)$ for each order, where c_1 and c_2 are the numbers of distinct n-grams of that order whose count is one and two (below the highest order, the continuation count).

The function below implements Equation 8.10. It counts the n-grams of the training tokens and derives the continuation counts of each lower order from the distinct n-grams one order up. It then computes $N(u)$, $S(u)$, and D for each order and evaluates the recursion for every token of a sequence, using as much context as precedes it. It returns the natural logarithm of each probability.

```
def kneser_ney(train, tokens, vocab_size, n=3):
    """Interpolated Kneser-Ney log-probabilities (nats) of each of tokens."""
    counts = {n: collections.Counter(zip(*(train[i:] for i in range(n))))}
    for k in range(n - 1, 0, -1):  # Lower orders: continuation counts
        counts[k] = collections.Counter(g[1:] for g in counts[k + 1])
    stats = {}
    for k, c in counts.items():
        c1, c2 = (sum(x == r for x in c.values()) for r in (1, 2))
        N, S = collections.Counter(), collections.Counter()
        for g, x in c.items():
            N[g[:-1]] += x
            S[g[:-1]] += 1
        stats[k] = c, N, S, c1 / max(c1 + 2 * c2, 1)
    logp = []
    for i in range(len(tokens)):
        p = 1 / vocab_size  # Below the unigram: uniform
        for k in range(1, min(n, i + 1) + 1):
            c, N, S, D = stats[k]
            g = tuple(tokens[i - k + 1:i + 1])
            if N[g[:-1]]:  # An unseen context passes p down unchanged
                p = (max(c[g] - D, 0) + D * S[g[:-1]] * p) / N[g[:-1]]
        logp.append(math.log(p))
    return logp
```

Before we evaluate the model, it is instructive to watch it at work on the one context we know well, “the time”. The cell below prints the discount for trigrams, and then the probabilities that Kneser–Ney and Laplace smoothing assign to three successors: “traveller”, which followed “the time” in 47 of its 87 occurrences, and two words that never followed it in training, “was” and “filby”.

```
c3 = collections.Counter(zip(words_train, words_train[1:], words_train[2:]))
c1, c2 = (sum(x == r for x in c3.values()) for r in (1, 2))
print(f'trigram discount D = {c1 / (c1 + 2 * c2):.2f}')
for w in ('traveller', 'was', 'filby'):
    lp = kneser_ney(words_train, ['the', 'time', w], vocab_size)[-1]
    print(f'P({w} | the time): Kneser-Ney {math.exp(lp):.2g}, '
          f'Laplace {models[3].prob(["the", "time"], w):.2g}')
```

```
trigram discount D = 0.93
P(traveller | the time): Kneser-Ney 0.53, Laplace 0.36
P(was | the time): Kneser-Ney 0.00037, Laplace 7.7e-05
P(filby | the time): Kneser-Ney 2.3e-05, Laplace 7.7e-05
```

The discount is about 0.9, so Kneser–Ney takes nearly a whole count from every observed trigram. That is how little a single observation is worth. A count of 47 hardly notices the loss, and "traveller" stays close to its relative frequency of 47/87, whereas Laplace smoothing gives away a third of that probability to the rest of the vocabulary. The unseen successors are more telling still. Laplace smoothing assigns "was" and "filby" the same probability, since it cannot tell a likely word from an unlikely one. Kneser–Ney can: "was" follows many different contexts and "filby" very few, and its estimates for the two differ by more than a factor of ten. How much is this worth on the held-out text?

```
for n in (1, 2, 3):
    kn = kneser_ney(words_train, words_val, vocab_size, n)
    print(f'{n}-gram held-out perplexity, Kneser-Ney: '
          f'{math.exp(-sum(kn) / len(kn)):7.1f}')
```

```
1-gram held-out perplexity, Kneser-Ney:   592.8
2-gram held-out perplexity, Kneser-Ney:   312.3
3-gram held-out perplexity, Kneser-Ney:   300.1
```

Kneser–Ney turns the ranking around. Its unigram is already better than Laplace's, and its bigram halves the unigram's perplexity. Its trigram is the best of all six models, because an unseen trigram now falls back on the bigram and unigram estimates instead of on a uniform guess. The trigram gains little over the bigram, however: most held-out trigrams are still unseen, so for most tokens the trigram model is the bigram model. The job of a language model is low perplexity on *unseen* text, and perplexity is our main metric for the neural models of the following sections.

Kneser–Ney has one limit, which it shares with every count model. It can move probability mass to unseen events, but it cannot know which of them are plausible. A count table treats every token as an unrelated symbol, so it cannot see that "cat" and "feline" should behave alike, or that "I want to eat, grandma" is benign while "I want to eat grandma" is not. Sharing statistical strength between related words and contexts requires learned representations, and these are where neural networks enter, in Section 8.4.

8.3.2.4 Bits per Byte

Perplexity has one serious flaw: it depends on what a token is. A character model chooses among 27 symbols per step and our word model among thousands, so their perplexities cannot be compared. Tokenization is a design choice (Section 8.2), and a metric for comparing models must not depend on it. The repair is simple: we change the denominator. Instead of dividing the total surprisal by the number of tokens, we divide it by the number of *bytes* of text, which depends on the text alone. Measuring surprisal in bits (base-2 logarithms) gives *bits per byte*,

$$\text{bpb} = \frac{1}{B}\sum_{t=1}^{T} -\log_2 P(x_t \mid x_{<t}), \tag{8.11}$$

where B is the length of the evaluated text in UTF-8 bytes. Bits per byte is a compression rate: a model at bpb bits per byte, combined with an arithmetic coder, compresses the text to a fraction $\text{bpb}/8$ of its raw size.

To see the two metrics side by side, we estimate the same model class on the same normalized training text under three tokenizations: characters, words, and a 1,024-token BPE (the tokenizer of Section 8.2, trained here on the normalized training text). The model is a trigram with Laplace smoothing, and in the last column with Kneser–Ney smoothing. A fair comparison needs one correction. The word model's <unk> stands for any word it cannot name, so a prediction of <unk> does not yet say which word the text contains. To describe the held-out text, the word model must still spell each unknown word out, and we add $\log_2 27$ bits for each of its letters and for the space that ends it: the cost of a uniform choice among the 27 symbols of the normalized text.

```
bpe = d2l.BPETokenizer(vocab_size=1024, pattern=d2l.BPETokenizer.GPT2_PATTERN)
bpe.train(text[:split])
streams = {'character': (list(text[:split]), list(text[split:]),
                         len(set(text))),
           'BPE (1,024)': (bpe.encode(text[:split]), bpe.encode(text[split:]),
                           bpe.vocab_size),
           'word': (words_train, words_val, vocab_size)}
n_bytes = len(text[split:].encode('utf-8'))
spell = sum((len(w) + 1) * math.log(27)  # Nats to spell out the <unk> words
            for w in text[split:].split() if w not in known)
print(f'{"tokenization":>12} {"|V|":>5} {"tokens":>7} {"ppl":>7} '
      f'{"bpb":>5} {"KN bpb":>6}')
for name, (train, val, m) in streams.items():
    nll = eval_nll(NGramLM(train, 3, m), val)
    nll_kn = -sum(kneser_ney(train, val, m))
    extra = spell if name == 'word' else 0
    print(f'{name:>12} {m:>5} {len(val):>7} {math.exp(nll / len(val)):>7.1f} '
          f'{(nll + extra) / math.log(2) / n_bytes:>5.2f} '
          f'{(nll_kn + extra) / math.log(2) / n_bytes:>6.2f}')
```

```
tokenization   |V|  tokens     ppl   bpb KN bpb
   character    27   17345     6.4  2.68   2.67
 BPE (1,024)  1024    5509   281.6  2.58   1.90
        word  4316    3373  2481.4  2.81   2.22
```

Judged by perplexity, the three Laplace-smoothed models differ by more than two orders of magnitude. Judged by bits per byte, they lie within a third of a bit of each other, and the character model, by far the best by perplexity, no longer comes first. At a fixed order n, coarser tokens condition on more text: two words of context reach about ten characters back, while two characters barely finish a syllable. The word model would come first, were it not for its unknown words. Spelling them out adds about 0.6 bits per byte, which leaves it last with Laplace smoothing. The BPE model needs no such help, since any text can be written in its byte-level tokens, and it compresses best. This is one more reason modern models use subword tokenization.

Kneser–Ney, in the last column, lowers the bits per byte of the BPE and word trigrams by more than half a bit, but barely changes the character trigram. With 27 characters, few character trigrams are unseen, so smoothing has little to repair. The BPE trigram with Kneser–Ney, at about 1.9 bits per byte, is the strongest count model of this section.

This is why modern language models are compared by bits per byte: a model with a 50,000-token vocabulary and one with a 200,000-token vocabulary have no common perplexity scale. Three magnitudes are worth remembering. Shannon estimated from human pre-

dictions that printed English carries about one bit per character (Shannon 1951). Strong neural language models compress diverse web text to below one bit per byte, while the count models of this section need between about 1.9 and 2.8.

8.3.3 Partitioning Sequences

The neural language models of the following sections read the BPE token ids of Section 8.2 in minibatches of fixed-length subsequences. We now build that pipeline.

Think of the corpus as one stream of token ids $x_1, x_2, x_3, \ldots$. An input window of length T that starts at position t holds $x_t, \ldots, x_{t+T-1}$. Its target is the same window shifted forward by one token, $x_{t+1}, \ldots, x_{t+T}$. While the model reads the input, it is trained to predict each next token, so one window supplies T next-token predictions. Every position of the corpus except the last T starts a window, and consecutive windows overlap in all but one token. Figure 8.5 shows the first two (input, target) pairs over the opening tokens of the corpus, its title line. At training time we draw minibatches of windows at random, which provides the randomness that stochastic gradient descent expects.

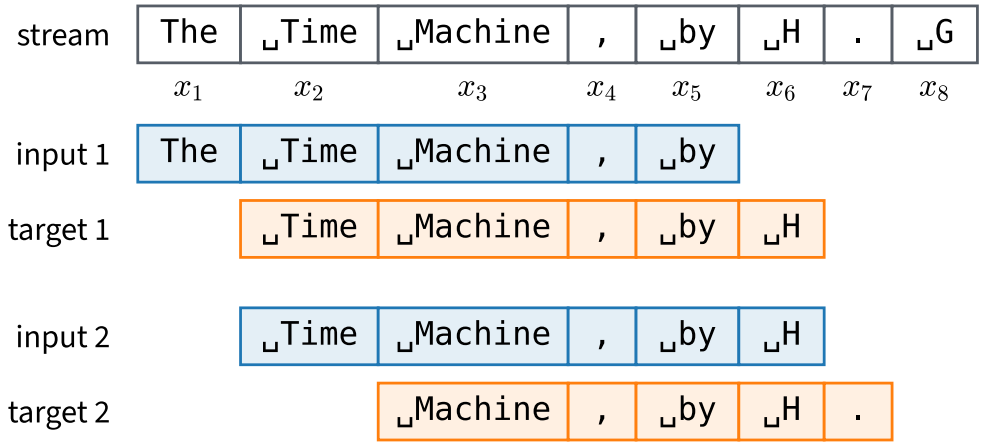


Figure 8.5: The first two input windows of length $T = 5$ over the opening BPE tokens of *The Time Machine* (␣ marks a leading space). Each target is its input shifted forward by one token, and consecutive windows overlap in all but one token; a minibatch gathers windows drawn at random.

The constructor below assembles the pipeline. It downloads the corpus, trains the tokenizer of Section 8.2 on it, and encodes the corpus into token ids. It then lays out all overlapping windows as rows of two aligned arrays, inputs `X` and targets `Y`. The window length T is the argument `num_steps`. The trained tokenizer remains available as `data.tokenizer` for decoding ids back into text, and `data.vocab` refers to it too, so that `len(data.vocab)` gives the vocabulary size.

```
@d2l.add_to_class(d2l.TimeMachine)
def __init__(self, batch_size, num_steps, num_train=50000, num_val=5000,
             tokenization='bpe', vocab_size=1024):
    super(d2l.TimeMachine, self).__init__()
```

```
        self.save_hyperparameters()
        raw_text = self._download()
        if tokenization == 'bpe':
            self.tokenizer = d2l.BPETokenizer(
                vocab_size, pattern=d2l.BPETokenizer.GPT2_PATTERN)
            self.tokenizer.train(raw_text)
            corpus, self.vocab = self.tokenizer.encode(raw_text), self.tokenizer
        else:  # tokenization='char': characters, via TimeMachine.build
            corpus, self.vocab = self.build(raw_text)
        windows = [corpus[i:i+num_steps+1] for i in range(len(corpus)-num_steps)]
        # Drop the windows that straddle the split, so that no validation token
        # is a training target
        windows = windows[:num_train] + windows[num_train+num_steps:]
        array = d2l.tensor(windows)
        self.X, self.Y = array[:,:-1], array[:,1:]
```

The first `num_train` windows are used for training and `num_val` later ones for validation. One detail needs care. Consecutive windows overlap, so the windows right after the training range contain tokens that training has already used as targets. The constructor therefore drops `num_steps` windows between the two ranges, and no validation token is a training target. Validation minibatches are read in order.

```
@d2l.add_to_class(d2l.TimeMachine)
def get_dataloader(self, train):
    idx = slice(0, self.num_train) if train else slice(
        self.num_train, self.num_train + self.num_val)
    return self.get_tensorloader([self.X, self.Y], train, idx)
```

Each minibatch is a pair of integer arrays of shape (`batch_size`, `num_steps`). Decoding one row checks that the ids are Wells' prose and that the target is the input shifted by one token.

```
data = d2l.TimeMachine(batch_size=2, num_steps=10)
for X, Y in data.train_dataloader():
    print('X:', X, '\nY:', Y)
    break
print('X[0]:', repr(data.tokenizer.decode([int(i) for i in X[0]])))
print('Y[0]:', repr(data.tokenizer.decode([int(i) for i in Y[0]])))
```

```
X: tensor([[ 10, 103, 262, 119, 307, 259, 824,  44, 269, 486],
        [572, 817, 317, 562, 277, 260, 890, 688, 669, 585]])
Y: tensor([[103, 262, 119, 307, 259, 824,  44, 269, 486, 260],
        [817, 317, 562, 277, 260, 890, 688, 669, 585, 256]])
X[0]: '\ngrew thinner, I saw'
Y[0]: 'grew thinner, I saw the'
```

A window offers far more context than a count table can use: our tables ran out of data at two words of context. The next section replaces the table with a neural network that carries a summary of the whole prefix.

8.3.4 Summary

A language model assigns probabilities to token sequences, and the chain rule turns this into next-token prediction, so one model both scores and generates text. n-gram models estimate the conditionals by counting under a Markov assumption. They are easy to estimate and write locally convincing text. But the number of possible contexts grows exponentially in n, and Zipf's law leaves most observed n-grams with a single count, so these models memorize rather than generalize. Perplexity, the exponentiated average cross-entropy, measures a model as an effective branching factor per token, and it shows that the fluent trigram with Laplace smoothing is the worst of our models on new text. Interpolated Kneser–Ney discounts every count and hands the freed mass to lower orders that count distinct predecessors. It makes the trigram the best count model, but like any smoothing it can only move probability mass to the zeros. Because perplexity is defined per token, it cannot compare models with different tokenizers, while bits per byte can. The training data for the rest of the chapter is one stream of BPE token ids, cut into overlapping windows whose targets are the inputs shifted by one token.

8.3.5 Exercises

1. code **Temperature sampling.** `NGramLM.sample` draws each next token in proportion to its raw count N. Add a temperature $\tau > 0$ by drawing in proportion to $N^{1/\tau}$ instead.
    1. Generate trigram text from the prefix `['the', 'time']` at $\tau \in \{0.3, 1, 3\}$ and describe what changes.
    2. Which distribution over successors does the sampler approach as $\tau \to 0$, and which as $\tau \to \infty$? For a context that occurred exactly once in training, what does the temperature change?

2. **Dense versus sparse storage.** Consider a 5-gram model over a vocabulary of $|\mathcal{V}| = 50{,}000$ word tokens.
    1. How many entries does a dense table of the conditional probabilities Equation 8.6 contain?
    2. Suppose instead that only the 5-grams observed in a trillion-token corpus are stored, at 16 bytes per entry. How much memory is needed at most, and what fraction of all possible 5-grams is covered?
    3. The word vocabulary of Section 8.3.1 has $|\mathcal{V}| = 4{,}316$ entries, and the code there prints the number of distinct word trigrams in the training text. Which fraction of all possible trigrams occurs in training? Which fraction of the held-out trigrams does? Is the coverage computed in (b) therefore a good estimate of how often a query on new text finds its 5-gram in the table?

3. code **Smoothing constant.** Sweep α in Equation 8.9 over the 19 values $\alpha = 10^{k/4}$ for $k = -16, -15, \dots, 2$, for the unigram, bigram, and trigram word models, and plot held-out perplexity against α on a logarithmic axis.
    1. How does the best α change with n, and why?
    2. Rank the three models, each at its own best α, by held-out perplexity. How does this ranking differ from the one printed in Section 8.3.2 with the default $\alpha = 0.01$ of `NGramLM`?

4. **Perplexity under skew.** A training corpus over ten symbols consists of one symbol 91% of the time and each of the other nine 1% of the time.

1. Compute the perplexity of the unigram model estimated from this corpus on a long test text drawn from the same distribution. How does it compare with the vocabulary size of ten?
2. The word-level unigram model of Section 8.3.2 chooses among thousands of word types. Read off its held-out perplexity as an effective number of equally likely choices, and explain with Zipf's law Equation 8.7 why this number is far below $|\mathcal{V}|$.

Adapted from Daniel Jurafsky and James H. Martin, Speech and Language Processing, 3rd ed. draft, Exercise 3.12.

5. **Bits per byte from perplexity.** A model achieves per-token perplexity PPL on a text that its tokenizer splits into T tokens and whose size is B UTF-8 bytes.

   1. Show that its bits per byte Equation 8.11 equals

   $$\text{bpb} = \frac{T}{B} \log_2 \text{PPL}.$$

   2. Use the formula and the table printed in Section 8.3.2.4 to compute the average number of bytes per token, B/T, for each tokenization. Explain how bits per byte that nearly agree can coexist with perplexities that differ by more than two orders of magnitude.
   3. What held-out perplexity would the BPE-level trigram need in order to reach one bit per byte?

6. **Overlapping windows.** `d2l.TimeMachine` builds one row of `num_steps + 1` consecutive tokens for every possible start position in the corpus. The first `num_steps` tokens of a row form a row of `X`, and the last `num_steps` tokens form the matching row of `Y`. The first `num_train` rows go to training, the next `num_steps` rows are dropped, and the `num_val` rows after them go to validation (Section 8.3.3). Take `num_steps=32`, `num_train=10000`, and `num_val=5000`.

   1. Counting input and target tokens together, which corpus positions do the training rows touch, and which do the validation rows touch? How many positions would both touch if no rows were dropped, and why would that inflate the validation score?
   2. How many times per epoch does a token in the interior of the training range serve as a target?
   3. How many windows does a partition of the entire corpus of about 66,000 tokens into non-overlapping 32-token windows yield? Compare this partition with the 10,000 overlapping training rows in the number of windows and in the number of distinct corpus positions covered.

7. code **Normalization across lengths.** `NGramLM` has no end-of-sequence token. Train a bigram `NGramLM` with $\alpha = 1$ and `vocab_size=3` on a short string over a three-symbol vocabulary. Use `prob` and the chain rule Equation 8.5 to sum the probabilities of all sequences of length 2, and separately those of all sequences of length 3.

   1. What does each sum equal, and what do the two sums say about the model as a distribution over sequences whose length is not fixed in advance?
   2. Add an end-of-sequence token to the vocabulary, so that `vocab_size=4`, and append it to the training string. Treat a sequence as complete once it emits this token. Show that the probabilities of all complete sequences now sum to exactly one.

Which property of the model rules out probability mass on sequences that never end?

Adapted from Daniel Jurafsky and James H. Martin, Speech and Language Processing, 3rd ed. draft, Exercise 3.5.

8. code **Seen and unseen trigrams.** ▯ Split the held-out positions from the third word on by whether the word trigram ending at that position occurred in training. Take the bigram and the trigram word models, each at its best α from the sweep of problem 3, and compute the average negative log-likelihood of each model on each group. On which group is the trigram the better model, and how does Kneser–Ney Equation 8.10 exploit this?

8.4 Recurrent Neural Networks

The language models of Section 8.3 predict each token from a short window of the tokens just before it. The trigram model, for example, sees only the two words before the one it predicts, and nothing older can influence the prediction. A human reader does better: a character introduced on the first page of a novel still shapes how its last page reads. How can a model take the whole prefix into account without growing as the prefix grows?

Widening the window is the obvious remedy, and it does not get far. A count table over a context of 20 BPE tokens has $1024^{20} \approx 10^{60}$ possible contexts, almost none of which any corpus contains. A neural network that reads the last 20 tokens fares better, since it can generalize between similar contexts. Its window, however, is still fixed, and a token 21 positions back is invisible.

Section 8.1 described the alternative: a *latent* model that summarizes the whole prefix in a *hidden state* $\mathbf{h}_{t-1}$, a vector of fixed size, and predicts from that summary,

$$P(x_t \mid x_{<t}) \approx P(x_t \mid \mathbf{h}_{t-1}). \tag{8.12}$$

After each token, the state is updated from the new input and the previous state:

$$\mathbf{h}_t = g(\mathbf{h}_{t-1}, x_t). \tag{8.13}$$

Each step then needs the same computation, however long the prefix. Note, however, that the approximation is exact only if $\mathbf{h}_{t-1}$ retains everything about the prefix that matters for predicting x_t. That is a great deal to ask. A state of fixed size cannot keep everything, so it must learn what to keep. A *recurrent neural network* (RNN) computes g with a neural network layer whose weights are the same at every step (Elman 1990). We define the RNN, turn it into a language model, and count that model's parameters and computation, before Section 8.5 implements and trains it.

A word of caution: "hidden" means something different here than in the hidden layers of Chapter 4. A hidden layer sits between the input and the output of one computation. A hidden state is carried from one time step to the next, and it depends on every input seen so far.

8.4.1 Neural Networks without Hidden States

A recurrent network is an MLP with one extra input, the previous hidden state. Let us therefore start from the MLP of Section 4.1, with one hidden layer and activation function ϕ. Its input is a minibatch of b examples with d features each, $\mathbf{X} \in \mathbb{R}^{b \times d}$. (We write b for the minibatch size, since n is the order of an n-gram in this chapter.) The hidden layer computes

$$\mathbf{H} = \phi(\mathbf{X}\mathbf{W}_{\text{xh}} + \mathbf{b}_{\text{h}}), \tag{8.14}$$

with weight $\mathbf{W}_{\text{xh}} \in \mathbb{R}^{d \times h}$, bias $\mathbf{b}_{\text{h}} \in \mathbb{R}^{1 \times h}$ (broadcast over the rows, Section 1.1.3.2), and h hidden units, so that $\mathbf{H} \in \mathbb{R}^{b \times h}$. The output layer computes

$$\mathbf{O} = \mathbf{H}\mathbf{W}_{\text{hq}} + \mathbf{b}_{\text{q}},$$

with $\mathbf{W}_{\text{hq}} \in \mathbb{R}^{h \times q}$ and $\mathbf{b}_{\text{q}} \in \mathbb{R}^{1 \times q}$. For classification, $\text{softmax}(\mathbf{O})$ gives a distribution over q classes.

To turn this MLP into a language model, we feed it the last m tokens and let it classify the next one, with one class per vocabulary token, $q = |\mathcal{V}|$. What should the input be? A token id will not do, since the ids are arbitrary labels. Instead each id selects one row of a learned table $\mathbf{E} \in \mathbb{R}^{|\mathcal{V}| \times d}$. This row is the token's *embedding*, a vector of d numbers. Because the table is learned, tokens that occur in similar contexts can receive similar vectors, and what the model learns about "cat" then transfers to "feline". The count tables of Section 8.3 had no means of doing this.

The m embeddings are concatenated into one input of length md. This *neural n-gram* model (Bengio et al. 2003) replaces the count table by a learned function, so a context that never occurred in training still gets a sensible prediction when it resembles contexts that did. The window remains, however. The model sees m tokens and nothing older, and its first weight matrix has md rows, so each additional token of context adds dh weights: 8,192 for $d = 64$ and $h = 128$.

8.4.2 Recurrent Neural Networks with Hidden States

An RNN gives the hidden layer one more input: its own output from the previous time step. At step t the input is a minibatch $\mathbf{X}_t \in \mathbb{R}^{b \times d}$, whose rows are the inputs of b sequences at that step, and the hidden state is $\mathbf{H}_t \in \mathbb{R}^{b \times h}$, one state $\mathbf{h}_t$ per row. A new weight $\mathbf{W}_{\text{hh}} \in \mathbb{R}^{h \times h}$ says how the previous state enters the current one:

$$\mathbf{H}_t = \phi(\mathbf{X}_t\mathbf{W}_{\text{xh}} + \mathbf{H}_{t-1}\mathbf{W}_{\text{hh}} + \mathbf{b}_{\text{h}}). \tag{8.15}$$

Compared with Equation 8.14, the only new term is $\mathbf{H}_{t-1}\mathbf{W}_{\text{hh}}$, and with it Equation 8.15 is an instance of Equation 8.13. Because $\mathbf{H}_t$ is computed from $\mathbf{H}_{t-1}$ by the same formula at every step, the computation is *recurrent*. A layer that computes Equation 8.15 is a *recurrent layer*, and a network with recurrent layers is a recurrent neural network. The state starts from $\mathbf{H}_0 = \mathbf{0}$. At each step the output layer reads the state,

$$\mathbf{O}_t = \mathbf{H}_t\mathbf{W}_{\text{hq}} + \mathbf{b}_{\text{q}}.$$

The same weights and biases, three arrays in the recurrent layer and two in the output layer, are used at every time step, so their number does not depend on the length of the sequence.

How does the state remember? Take the smallest case: one input and one hidden unit ($d = h = 1$), $\phi = \tanh$, $w_{\text{xh}} = 1$, $w_{\text{hh}} = 0.5$, no bias, and the input sequence $1, 0, 0$. The state is $\tanh(1) = 0.76$ after the first step, $\tanh(0.5 \cdot 0.76) = 0.36$ after the second, and $\tanh(0.5 \cdot 0.36) = 0.18$ after the third. The first input still shapes the state two steps later, scaled by about w_{hh} at every step. With $w_{\text{hh}} = 0$ it would be gone after one step. The recurrent weight sets how long the past persists.

Can a state also do what no window can? Consider a sequence of bits, and ask at every step whether the number of ones so far is even or odd. A model that reads the last m inputs must fail, whatever the value of m: flipping a bit that has left the window flips the answer and leaves the model's input unchanged. A state needs to hold a single bit, the parity so far, and to flip it whenever the input is 1. Two hidden units with a threshold activation suffice, for sequences of any length (the last exercise asks for their weights).

Figure 8.6 unrolls an RNN across three adjacent time steps. The horizontal arrows carry the state forward, and the weight labels repeat at every step because the parameters are shared. Unrolled over T steps, an RNN is a network T layers deep whose layers all have the same weights.

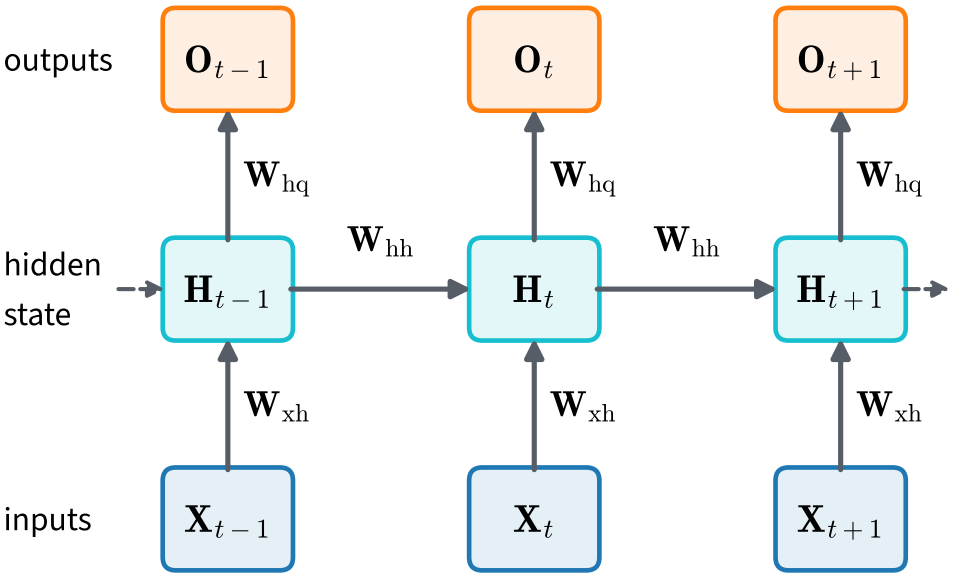


Figure 8.6: An RNN with a hidden state, unrolled across three adjacent time steps. The same weights are reused at every step.

The two products in Equation 8.15 can also be written as one, by concatenating input and state along the columns and stacking the two weight matrices along the rows:

$$[\mathbf{X}_t, \mathbf{H}_{t-1}] \begin{bmatrix} \mathbf{W}_{\text{xh}} \\ \mathbf{W}_{\text{hh}} \end{bmatrix} = \mathbf{X}_t \mathbf{W}_{\text{xh}} + \mathbf{H}_{t-1} \mathbf{W}_{\text{hh}}. \tag{8.16}$$

So the recurrent layer is nothing more than a fully connected layer applied to the concatenation of input and state. Current libraries nonetheless keep the two products separate: during training the inputs of all steps are known in advance, so their product with $\mathbf{W}_{\text{xh}}$ can be computed in one multiplication, outside the loop over time steps (exercise 2).

8.4.2.1 Constant Memory per Step

To compute $\mathbf{H}_t$, the recurrence needs only $\mathbf{H}_{t-1}$ and $\mathbf{X}_t$. Everything older can be discarded once $\mathbf{H}_t$ is formed, so memory and computation per step stay constant however long the sequence grows. For the model that Section 8.5 trains, with $h = 128$ hidden units stored as 32-bit floats, the state of one sequence occupies 512 bytes, whether the prefix holds ten tokens or a million.

The constant size has a drawback. The state is a *lossy* summary of the past. No novel fits into 512 bytes, so to record something new the network must overwrite something old. The opposite design keeps a representation of every past token and lets the model look back over all of them when it predicts. This is *attention* (Chapter 10), whose memory and computation per step grow with the length of the sequence.

8.4.3 RNN Language Models

An RNN language model reads one token per step and predicts the next. Its input layer is the embedding table of the neural n-gram: at step t, the token ids of the b sequences select b rows of $\mathbf{E}$, which form $\mathbf{X}_t \in \mathbb{R}^{b \times d}$. Selecting a row is the same as multiplying a one-hot vector of length $|\mathcal{V}|$ by $\mathbf{E}$, but the lookup never forms that vector. The recurrent layer updates the state, and the output layer maps $\mathbf{H}_t$ to $q = |\mathcal{V}|$ logits per sequence. A softmax over the logits gives $P(x_{t+1} \mid x_{\le t})$, the model's distribution over the next token. Because $\mathbf{H}_t$ depends on the entire prefix, the model approximates every conditional of the factorization $P(x_1, \dots, x_T) = \prod_t P(x_t \mid x_{<t})$ (Equation 8.5 in Section 8.3) without a Markov cutoff.

Figure 8.7 shows how such a model is trained on the data of Section 8.3.3. *The Time Machine* contains the phrase "thing that struck me", which the BPE tokenizer of Section 8.2 splits into the five tokens `␣thing`, `␣that`, `␣stru`, `ck`, and `␣me` (␣ marks a leading space; "struck" is not in the 1,024-token vocabulary and becomes two tokens). The input is the first four tokens, and the target is the same stream shifted forward by one, `␣that`, `␣stru`, `ck`, `␣me`. At every step, the cross-entropy loss compares the softmax of the output with the target, and the losses are averaged over the steps. At step 3, for instance, the model has read `␣thing ␣that ␣stru` and must predict `ck`, completing the word it has started. The average loss is the logarithm of the perplexity of Section 8.3, so minimizing it minimizes perplexity.

8.4.3.1 Model Size and Computation

Where do the parameters of an RNN language model sit? We count them for the first model that Section 8.5 trains on *The Time Machine*: the BPE vocabulary has $|\mathcal{V}| = 1{,}027$ ids (1,024 tokens and three special tokens), the embeddings have $d = 64$ dimensions, and the state has $h = 128$ units.

part	shape	parameters
embedding table $\mathbf{E}$	$1{,}027 \times 64$	65,728
input weight $\mathbf{W}_{\text{xh}}$	64×128	8,192
recurrent weight $\mathbf{W}_{\text{hh}}$	128×128	16,384

part	shape	parameters
hidden bias $\mathbf{b}_{\mathrm{h}}$	128	128
output weight $\mathbf{W}_{\mathrm{hq}}$	$128 \times 1{,}027$	131,456
output bias $\mathbf{b}_{\mathrm{q}}$	$1{,}027$	1,027
total		222,915

The recurrent layer, which does all the remembering, holds only 24,704 parameters, 11% of the total. The other 89% sit in the embedding table and the output layer, whose sizes scale with the vocabulary. Computation per token is split the same way. The recurrent layer takes $64 \cdot 128 + 128 \cdot 128 = 24{,}576$ multiply-adds, the output layer $128 \cdot 1{,}027 = 131{,}456$, more than five times as many, and the embedding lookup only copies a row. A larger vocabulary therefore makes the model larger and slower without enlarging the state. A larger state holds more of the past, and its recurrent weight grows with h^2.

8.4.3.2 Teacher Forcing

During training, the input at each step is the *true* previous token, taken from the corpus, whatever the model predicted at the step before. Training a sequence model on the true prefixes in this way is called *teacher forcing*. Every step is conditioned on a correct history, and all the targets of a window are known in advance, so the loss of the whole window is computed in one pass over it.

At generation time there are no true tokens to feed. We give the model a short prompt, draw a token from its next-token distribution (or take the most probable one), and feed that token back in as the next input. The model is now conditioned on its *own* outputs. This is the gap between one-step and multistep prediction that Section 8.1 measured on a sine wave. A token the model gets slightly wrong changes the state it continues from, and errors can compound over a long generation. Teacher forcing never shows the model its own mistakes, and Section 8.7 studies generation, where the model has to live with them.

Two things remain open. The first is a precaution. The loss at step t depends on the weights through every earlier step of the recurrence, so its gradient contains products of one factor per step, and such products tend to vanish or explode as the number of steps grows. Section 8.5 therefore limits the size of the gradient when it trains the model, and Section 8.6 analyzes the products themselves. The second is a question. The state *may* depend on the entire prefix, but nothing in the definition says how much of the prefix a trained state will hold. That can only be measured.

8.4.4 Summary

A recurrent neural network computes its hidden state from the current input and the previous state with the same weights at every step. Its parameter count does not depend on the sequence length, and its state is a fixed-size, lossy summary of the whole past, so memory and computation per step stay constant. An RNN language model embeds each token, updates the state, and maps it to logits over the vocabulary. Most of its parameters and computation sit in the embedding table and the output layer, which scale with the vocabulary. It is trained by teacher forcing to predict each next token from the true prefix, which minimizes perplexity, while generation feeds back its own outputs.

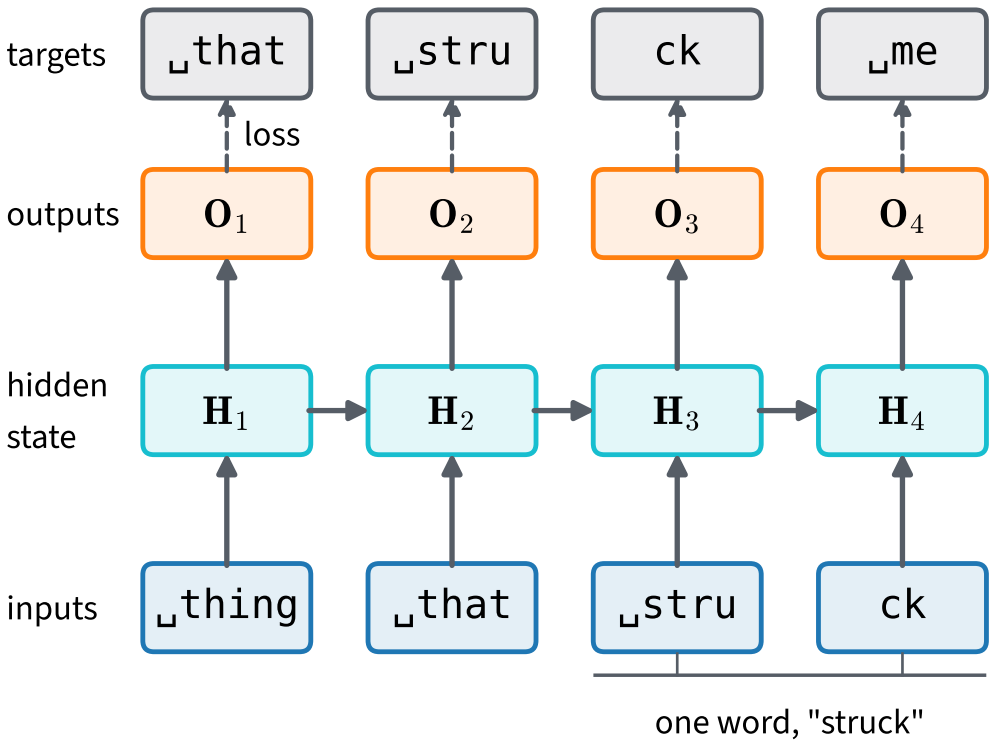


Figure 8.7: An RNN language model trained by teacher forcing over BPE tokens of *The Time Machine*. The target sequence is the input shifted forward by one token, so each step is trained to predict the next token.

8.4.5 Exercises

1. **Parameter counts.** Consider next-token prediction over a vocabulary $\mathcal{V}$ from a context of m previous tokens. Evaluate every count below for $|\mathcal{V}| = 10{,}000$, $m = 20$, and $h = d = 256$.
   1. An $(m+1)$-gram stores a count for every possible context and successor. How many entries does its table hold, and by what factor does that number grow when m increases by one?
   2. The RNN language model of this section has h hidden units, an embedding table of d-dimensional vectors, and an output layer with one logit per vocabulary token. Count its parameters, embedding table included. How does the count depend on m?
   3. The neural n-gram of Bengio et al. (2003) feeds the concatenated embeddings of the m context tokens to a hidden layer of width h, followed by the same output layer. Count its parameters as a function of m, embedding table included, and state how many parameters each additional context token adds. Compare the growth in m of the three models.
2. **Separating the input product.** Let $\mathbf{X}_t \in \mathbb{R}^{b\times d}$ and $\mathbf{H}_{t-1} \in \mathbb{R}^{b\times h}$ be the input and the state in Equation 8.15.
   1. During training, the inputs $\mathbf{X}_1, \ldots, \mathbf{X}_T$ of a window are all known before the first time step is computed. Which of the two products $\mathbf{X}_t\mathbf{W}_{\text{xh}}$ and $\mathbf{H}_{t-1}\mathbf{W}_{\text{hh}}$ can be computed for all T steps in a single matrix multiplication, and what are the shapes of its factors? Which product must stay inside the sequential loop over time steps, and why?
   2. Compare the work per step left inside the loop with the work per step of the concatenated form $[\mathbf{X}_t, \mathbf{H}_{t-1}]$. During generation, the input at step t is the token generated at step $t-1$. Does the advantage of the separate form survive there?
3. **Exposure bias.** Teacher forcing (Section 8.4.3.2) conditions every training step on the true prefix, whereas generation conditions each step on the model's own earlier outputs. The resulting mismatch is called *exposure bias* (Ranzato et al. 2016).
   1. Suppose generation samples each token from the model's next-token distribution, and the model assigns probability $1-\epsilon$ to the correct next token whenever its prefix is correct. What is the probability that T generated tokens contain no error? Evaluate it for $\epsilon = 0.01$ and $T = 100$.
   2. *Scheduled sampling* (Bengio et al. 2015) exposes the model to its own outputs during training: at each step, with some probability, the input is a token sampled from the model's prediction at the previous step instead of the true token. Which property of teacher forcing that Section 8.4.3.2 relies on does this give up, and how does that change the computation of a training window?
4. **A window inside a recurrence.** The neural n-gram reads a fixed window of the last m input embeddings. Take ϕ to be the identity and $\mathbf{b}_{\text{h}} = \mathbf{0}$ in Equation 8.15, and give the state $h = md$ units.
   1. Choose $\mathbf{W}_{\text{xh}}$ and $\mathbf{W}_{\text{hh}}$ so that, starting from $\mathbf{H}_0 = \mathbf{0}$, the state is the concatenation $\mathbf{H}_t = [\mathbf{X}_t, \mathbf{X}_{t-1}, \ldots, \mathbf{X}_{t-m+1}]$ for every $t \geq m$.
   2. Show that $\mathbf{W}_{\text{hh}}^m = \mathbf{0}$, and explain why this means that no input older than m steps affects $\mathbf{H}_t$.
   3. For $d = 256$ and $m = 20$, how many state units does this construction need? Explain what an RNN whose state has only 256 units must do instead of storing the window.
5. code **Embedding lookup as a product.** Let $\mathbf{E} \in \mathbb{R}^{|\mathcal{V}|\times d}$ be an embedding table, and let

x hold the token ids of a minibatch of b tokens.

1. Compute the inputs once by selecting the rows of **E** that **x** indexes and once as the product of the $b \times |\mathcal{V}|$ one-hot matrix of **x** with **E**, and check that the two agree. Time both for $|\mathcal{V}| \in \{10^3, 10^4, 10^5\}$ with $d = b = 256$, and state how the running time of each grows with $|\mathcal{V}|$.
2. Differentiate the sum of all entries of the selected rows with respect to **E** by automatic differentiation. Which rows of the gradient are nonzero, and what does a row contain when its token occurs several times in the minibatch? For $|\mathcal{V}| = 10^4$ and $b = 256$ ids drawn uniformly at random, estimate the fraction of rows that receive a nonzero gradient.

6. **Parity beyond any window.** ▣ Let $x_1, x_2, \ldots$ be a sequence of bits, and let the desired output at step t be the parity of $x_1 + \cdots + x_t$.
   1. Show that no model that reads only the last m inputs can produce the correct output at every step of a sequence longer than m, whatever the finite m.
   2. Let ϕ be a threshold activation. Construct an RNN of the form Equation 8.15 with two hidden units whose output equals the parity at every step, for any sequence length. Give the weights, and show that one hidden unit does not suffice.
   3. How many bits of information about $x_1, \ldots, x_t$ must the hidden state retain for the outputs at all later steps to be correct? Relate this to the description of the state as a lossy fixed-size summary in Section 8.4.2.1.

8.5 Implementing RNNs

Section 8.4 wrote down a recurrent language model and counted its parameters, but did not train it. Two questions are open: how its equations become working code, and what its hidden state is worth on *The Time Machine*, measured against the counts of Section 8.3. The first is the easier one. The model fits in three equations: look up the embedding of the current token, update the hidden state from it and from the previous state, and map the new state to scores over the vocabulary. The code for them has a simple interface. At each time step the model consumes one token per sequence and the previous hidden state, and it returns $|\mathcal{V}|$ logits per sequence and a new state of the same shape as the old one. Because the shapes do not change from step to step, the same step applies to a sequence of any length.

Let us first write this step with plain tensor operations, so that every parameter and every shape is visible, and then hand it to the recurrent layer that each framework provides. Training needs one addition that the equations do not show. The gradient passes through the same transition once per time step and can become large enough to undo much of the progress of training in a single update. We therefore clip its norm before every update, a crude remedy and a standard one, and we train once without it to see what it prevents. The second question we answer by measurement. The trained model is scored against the count-based n-gram models on the same text, token for token, and then position by position, to find out how much of what it has read the model uses. Finally we let it generate text.

8.5.1 An RNN Language Model from Scratch

The model has three parts, and we implement each as its own class or method: a *recurrent cell* that updates the hidden state one time step at a time, an embedding that turns token ids into the vectors the cell consumes, and an output layer that maps each hidden state to a vector of logits over the vocabulary.

8.5.1.1 The Recurrent Cell

For a minibatch of b sequences, let $\mathbf{X}_t \in \mathbb{R}^{b\times d}$ hold their inputs at step t and $\mathbf{H}_t \in \mathbb{R}^{b\times h}$ their hidden states. The cell computes the recurrence of Section 8.4 with the tanh activation,

$$\mathbf{H}_t = \tanh(\mathbf{X}_t\mathbf{W}_{\text{xh}} + \mathbf{H}_{t-1}\mathbf{W}_{\text{hh}} + \mathbf{b}_{\text{h}}), \tag{8.17}$$

so each new state mixes the current input with the previous state through two weight matrices, $\mathbf{W}_{\text{xh}} \in \mathbb{R}^{d\times h}$ and $\mathbf{W}_{\text{hh}} \in \mathbb{R}^{h\times h}$, and a bias $\mathbf{b}_{\text{h}} \in \mathbb{R}^{h}$. The `RNNScratch` class holds these three parameters. The number of hidden units `num_hiddens` (h) is a hyperparameter, and `num_inputs` (d) is the width of the vectors arriving at each step, which will shortly be the embedding dimension.

As in Section 2.4, we register the weights with the framework's parameter type, so that the optimizer finds them and the trainer can move them to a GPU. Both weight matrices start from $\mathcal{N}(0, \sigma^2)$ with $\sigma = 0.01$, and the bias starts at zero. With weights this small, the largest eigenvalue of the recurrent weight matrix has magnitude about $\sigma\sqrt{h} \approx 0.11$. Recall the scalar example of Section 8.4, where the past persisted scaled by about w_{hh} per step: at 0.11 per step, the untrained cell forgets its state within a few steps. Training has to grow the weights from there.

```
class RNNScratch(d2l.Module):
    """The RNN model implemented from scratch."""
    def __init__(self, num_inputs, num_hiddens, sigma=0.01):
        super().__init__()
        self.save_hyperparameters()
        self.W_xh = nn.Parameter(
            d2l.randn(num_inputs, num_hiddens) * sigma)
        self.W_hh = nn.Parameter(
            d2l.randn(num_hiddens, num_hiddens) * sigma)
        self.b_h = nn.Parameter(d2l.zeros(num_hiddens))
```

The forward computation loops over the outermost axis of `inputs`, whose shape is (`num_steps`, `batch_size`, `num_inputs`), applies the recurrence once per time step, and collects the hidden state of every step. When no initial state is supplied, the state starts at zero. The explicit Python loop over time steps is the recurrence as we wrote it on paper, which is all we ask of it for now.

```
@d2l.add_to_class(RNNScratch)
def forward(self, inputs, state=None):
    if state is None:
        # Initial state with shape: (batch_size, num_hiddens)
        state = d2l.zeros((inputs.shape[1], self.num_hiddens),
                          device=inputs.device)
    outputs = []
    for X in inputs:  # Shape of inputs: (num_steps, batch_size, num_inputs)
        state = d2l.tanh(d2l.matmul(X, self.W_xh) +
                         d2l.matmul(state, self.W_hh) + self.b_h)
        outputs.append(state)
    return outputs, state
```

We feed the cell a minibatch of two sequences of 100 steps each.

```
batch_size, num_inputs, num_hiddens, num_steps = 2, 16, 32, 100
rnn = RNNScratch(num_inputs, num_hiddens)
X = d2l.ones((num_steps, batch_size, num_inputs))
outputs, state = rnn(X)
```

We expect one state per step, each with the shape (`batch_size`, `num_hiddens`) of the state the cell started from. The helper below checks a shape; later sections use it as well.

```
def check_shape(a, shape):
    """Check the shape of a tensor."""
    assert a.shape == shape, \
            f'tensor\'s shape {a.shape} != expected shape {shape}'

check_shape(d2l.stack(outputs, 0), (num_steps, batch_size, num_hiddens))
check_shape(state, (batch_size, num_hiddens))
```

8.5.1.2 The Embedding Table

The cell consumes vectors, but our data pipeline delivers integer token ids. As in Section 8.4, an embedding lookup bridges the gap: a trainable table $\mathbf{E} \in \mathbb{R}^{|\mathcal{V}| \times d}$ represents token i by its i^{th} row, a d-dimensional vector that gradient descent adjusts along with every other parameter. Older treatments used a *one-hot encoding* instead: token i becomes the length-$|\mathcal{V}|$ vector with a single 1 in position i, and the input weights multiply that vector. Since multiplying a one-hot vector by a matrix selects a row of it (Section 8.4), this is nothing but a lookup, computed the expensive way. With a vocabulary of 27 characters nobody minds. With our BPE vocabulary of 1,027 ids, each step would multiply a vector that is zero everywhere but in one entry.

The `RNNLMScratch` class assembles the language model around a cell `rnn`: the embedding table `E` on the input side and an output layer (`W_hq`, `b_q`) on the output side. The embedding dimension is read off the cell's `num_inputs`, since the cell consumes what the embedding produces. The constructor also takes two dropout rates. Both are zero for now, and we return to them when the model overfits. The embedding rows start at unit scale, as the 1s of a one-hot encoding would, while the output weights start at scale σ and the output bias at zero. The logits of the untrained model are then close to zero, its next-token distribution is close to uniform, and its perplexity is close to $|\mathcal{V}|$. This is a prediction we can check before training.

The model inherits from `Classifier` the cross-entropy loss, averaged over all tokens of a minibatch. Its validation step returns only this loss, since the token accuracy that `Classifier` would add is not what we monitor. The mean cross-entropy is the logarithm of the perplexity (Section 8.3.2), so the validation perplexity is the exponential of the trainer's mean validation loss.

```
class RNNLMScratch(d2l.Classifier):
    """The RNN-based language model implemented from scratch."""
    def __init__(self, rnn, vocab_size, lr=0.01, dropout=0, emb_dropout=0):
        super().__init__()
        self.save_hyperparameters()
```

```
        self.sigma = getattr(rnn, 'sigma', 0.01)  # Scale of the output weights
        self.init_params()

    def init_params(self):
        self.E = nn.Parameter(
            d2l.randn(self.vocab_size, self.rnn.num_inputs))
        self.W_hq = nn.Parameter(
            d2l.randn(self.rnn.num_hiddens, self.vocab_size) * self.sigma)
        self.b_q = nn.Parameter(d2l.zeros(self.vocab_size))

    def validation_step(self, batch):
        return self.loss(self(*batch[:-1]), batch[-1])
```

The minibatches sampled by our data pipeline have shape (`batch_size`, `num_steps`). The `embedding` method looks up each id and transposes the result to (`num_steps`, `batch_size`, `num_inputs`), the layout whose outermost axis the cell's forward loop walks over.

```
@d2l.add_to_class(RNNLMScratch)
def embedding(self, X):
    # Output shape: (num_steps, batch_size, num_inputs)
    return self.E[X.T]
```

8.5.1.3 The Output Layer

The last of the three equations turns each hidden state into scores for the next token. The output layer computes $\mathbf{O}_t = \mathbf{H}_t \mathbf{W}_{\text{hq}} + \mathbf{b}_{\text{q}}$ for every time step, with $\mathbf{W}_{\text{hq}} \in \mathbb{R}^{h \times |\mathcal{V}|}$. Since the same weights serve every step, one matrix product handles all of them: the method stacks the states of all steps into a (`batch_size`, `num_steps`, `num_hiddens`) tensor and multiplies it by $\mathbf{W}_{\text{hq}}$, which gives logits of shape (`batch_size`, `num_steps`, `vocab_size`) that line up with the target array. The `forward` method chains the three parts together.

```
@d2l.add_to_class(RNNLMScratch)
def output_layer(self, rnn_outputs):
    # A list of per-step states, or a tensor whose first axis is time
    H = d2l.stack(list(rnn_outputs), 1)  # (batch_size, num_steps, num_hiddens)
    return d2l.matmul(H, self.W_hq) + self.b_q

@d2l.add_to_class(RNNLMScratch)
def forward(self, X, state=None):
    embs = F.dropout(self.embedding(X), self.emb_dropout, self.training)
    rnn_outputs, _ = self.rnn(embs, state)
    H = d2l.stack(list(rnn_outputs), 0)  # (num_steps, batch_size, num_hiddens)
    return self.output_layer(F.dropout(H, self.dropout, self.training))
```

A minibatch of token ids of shape (`batch_size`, `num_steps`) should come out as one vector of logits per position. We check this with a vocabulary of 1,024 ids, about the size of the one we train on.

```
model = RNNLMScratch(rnn, vocab_size=1024)
outputs = model(d2l.ones((batch_size, num_steps), dtype=d2l.int64))
check_shape(outputs, (batch_size, num_steps, 1024))
```

8.5.2 Gradient Clipping

A recurrent network is deep in time. Backpropagating through T time steps multiplies T Jacobians of the same transition, one per step, and as in a deep feedforward network such a product can shrink or grow exponentially with its length. Section 8.6 analyzes both cases. Vanishing gradients limit what the model can learn, but they do not break training. Exploding gradients can: when the product grows, an occasional gradient is many times longer than the typical one. Below, we train the same model with a remedy for them and without it.

To see what a large gradient threatens, consider the length of one update. A gradient step with learning rate η moves the parameters $\mathbf{w}$ to $\mathbf{w} - \eta\mathbf{g}$, a distance

$$\|(\mathbf{w} - \eta\mathbf{g}) - \mathbf{w}\| = \eta\|\mathbf{g}\|. \tag{8.18}$$

Thus the length of a step grows with the norm of the gradient. Gradient descent works because, over a short step, the loss is close to its linear approximation $f(\mathbf{w}) - \eta\|\mathbf{g}\|^2$, which decreases. When $\|\mathbf{g}\|$ spikes, the step leaves the region where that approximation holds, and the loss can jump instead of falling. One such step can undo the progress of many updates. What can we do about it? Shrinking η until even the rare spike is safe would slow every step to protect against a few. *Gradient clipping* leaves η alone and instead caps the norm on the occasions it is too large, projecting $\mathbf{g}$ onto the ball of radius θ:

$$\mathbf{g} \leftarrow \min\left(1, \frac{\theta}{\|\mathbf{g}\|}\right)\mathbf{g}. \tag{8.19}$$

A gradient with norm at most θ passes unchanged; a longer one is scaled down to norm θ and keeps its direction. With $\theta = 1$, a gradient of norm 10 becomes one tenth of itself. The norm $\|\mathbf{g}\|$ is the Euclidean norm of the gradients of all parameters of the model, stacked into one vector.

Note that this is a rather crude device. Whenever it acts, clipping changes the update, so training no longer follows the gradient, and its effect on convergence is hard to analyze. It is standard nevertheless: recurrent networks and large transformers alike are trained with it.

Clipping belongs to the trainer. Whenever a positive `gradient_clip_val` is set, `d2l.Trainer` (Section 2.4) clips the gradients after it has computed them and before the parameter update.

The method below rescales each parameter's gradient in place.

```
@d2l.add_to_class(d2l.Trainer)
def clip_gradients(self, grad_clip_val, model):
    params = [p for p in model.parameters() if p.requires_grad]
    norm = torch.sqrt(sum(torch.sum(p.grad ** 2) for p in params))
```

```
    scale = torch.clamp(grad_clip_val / norm, max=1.0)  # no `if` on the host
    for param in params:
        param.grad.mul_(scale)
```

8.5.3 Training

The `d2l.TimeMachine` pipeline of Section 8.3 downloads the corpus, trains the BPE tokenizer on it, and serves minibatches of token-id windows together with their shift-by-one targets. The corpus is 65,741 BPE tokens long, and training window i starts at token i. The pipeline drops the 32 windows that straddle the split, so that no validation token is ever a training target. By default the pipeline trains on the first 50,000 windows, most of the novella, and validates on the following 5,000.

```
data = d2l.TimeMachine(batch_size=1024, num_steps=32)
```

Every window is $T = 32$ tokens long, and every forward pass starts from a zero state. Neither information nor gradient crosses a window boundary, so gradients flow backward through at most 32 steps: the chain of dependencies is *truncated*. Whatever the novella does over longer ranges, the model can learn it only insofar as it helps within a window. Section 8.6 examines what this compromise gives up, and what changes when the state is carried from one window into the next.

8.5.3.1 A Plain Model

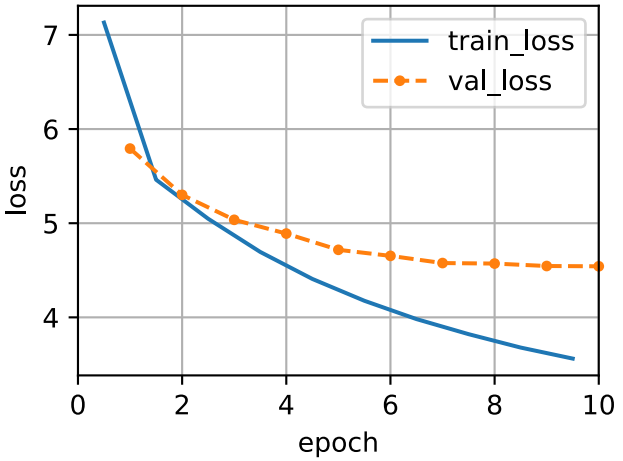
We first train the model as built so far, the one whose parameters Section 8.4 counted: an embedding of dimension $d = 64$ and a cell with $h = 128$ hidden units, for ten epochs of minibatch SGD at learning rate 4, clipping gradients to norm $\theta = 1$, with no dropout. A learning rate of 4 is large by the standards of the earlier chapters. Note what it means together with clipping: whenever clipping acts, the update has length $\eta\theta = 4$, whatever the gradient was. As in Section 8.4, training is teacher-forced, so every step conditions on the true prefix from the corpus. We keep a copy of the embedding table, to see later how much training changes it.

```
rnn = RNNScratch(num_inputs=64, num_hiddens=128)
model = RNNLMScratch(rnn, vocab_size=len(data.vocab), lr=4)
```

```
E_init = d2l.numpy(model.E).copy()
```

```
trainer = d2l.Trainer(max_epochs=10, gradient_clip_val=1, num_gpus=1)
ppl_init = math.exp(trainer.evaluate(model, data)['loss'])
print(f'initial validation perplexity {ppl_init:.0f}')
trainer.fit(model, data)
```

```
initial validation perplexity 1027
```



Before training, the validation perplexity is close to the vocabulary size of 1,027, as predicted for a model that knows nothing yet. The board plots the mean cross-entropy per token, the logarithm of the perplexity. Both curves fall fast during the first epochs. Then they separate: the training loss keeps falling, while the validation loss stops improving over the last few epochs. After ten epochs the training perplexity is less than half the validation perplexity. This should not surprise us. With 222,915 parameters for 50,000 training tokens, more than four parameters per token, the model has started to memorize its training text. The final validation perplexity is the exponential of the last validation loss.

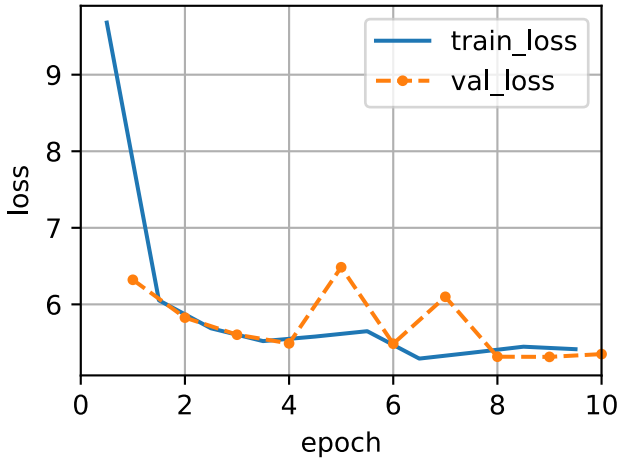
```
ppl_plain = math.exp(trainer.history['val_loss'][-1])
print(f'validation perplexity {ppl_plain:.1f}')
```

```
validation perplexity 93.9
```

How much of this result do we owe to clipping? We called it a crude device, so let us see what happens without it. We train a second copy of the plain model with the same settings, except that the gradients are not clipped, and print the highest validation perplexity of the run next to the final one.

```
unclipped = RNNLMScratch(RNNScratch(num_inputs=64, num_hiddens=128),
                         vocab_size=len(data.vocab), lr=4)
```

```
trainer_unclipped = d2l.Trainer(max_epochs=10, num_gpus=1)  # No clipping
trainer_unclipped.fit(unclipped, data)
ppls = [math.exp(loss) for loss in trainer_unclipped.history['val_loss']]
print(f'validation perplexity without clipping: highest {max(ppls):.0f}, '
      f'final {ppls[-1]:.0f}')
```




```
validation perplexity without clipping: highest 655, final 211
```

Without clipping, training is erratic. At some epoch the loss jumps instead of falling, as the argument for clipping warned, and the validation perplexity leaps into the hundreds or thousands, in some runs into the millions. Some runs recover part of the way and an occasional one never does, but every run ends at more than twice the perplexity of the clipped model, and most end far higher. Crude or not, clipping is what makes a learning rate of 4 usable.

A perplexity of about 90 to 100 may look poor next to the perplexity of about 6 of the character-level trigram in Section 8.3. The two numbers cannot be compared: our model chooses among 1,027 BPE tokens per step, a character model among 27 characters. Bits per byte (Section 8.3.2.4, Equation 8.11) removes the dependence on the tokenizer: it divides the total surprisal of a text by its length in bytes, which here comes to $\log_2$ ppl divided by the average number of bytes per token. We measure the bytes per token on the 5,000 consecutive validation tokens that start the validation windows.

```
ids = d2l.numpy(data.X[data.num_train:data.num_train+data.num_val, 0]).tolist()
bytes_per_token = len(data.tokenizer.decode(ids).encode('utf-8')) / len(ids)
print(f'{bytes_per_token:.2f} bytes/token, '
      f'{math.log2(ppl_plain) / bytes_per_token:.2f} bits per byte')
```

```
2.78 bytes/token, 2.36 bits per byte
```

Each token covers about 2.8 bytes of text, and the plain model needs about 2.3 to 2.4 bits per byte.

8.5.3.2 Embeddings under SGD

Overfitting is one problem of the plain model. A second one is hidden in the embedding table, and it takes a measurement to find it: how far did each row of **E** move during training, relative to its initial length?

```
E_final = d2l.numpy(model.E)
```

```
change = (((E_final - E_init) ** 2).sum(1) / (E_init ** 2).sum(1)) ** 0.5
print(f'mean change of an embedding row: {change.mean():.1%} of its length')
```

```
mean change of an embedding row: 4.2% of its length
```

The rows barely move. Each row receives a gradient only from the positions where its token occurs, and the loss averages over all $1{,}024 \times 32 = 32{,}768$ positions of a minibatch, so a token that occurs a few times contributes a few parts in 32,768 of the gradient. A step of SGD moves every parameter by the learning rate times its gradient, the same rate for all, which is far too small for rows that start at unit scale. Rows of tokens that never occur in the training text do not move at all. This is an awkward finding. The table that was meant to let the model choose a vector for each token is, in this configuration, little more than a fixed random projection.

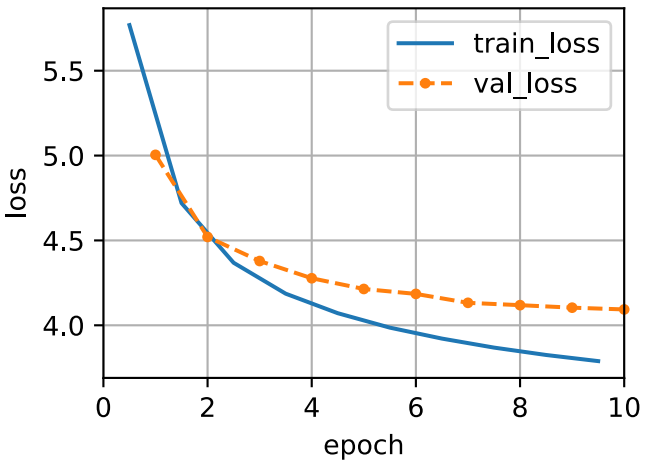
We do not cure this here. We work around it. The input weights $\mathbf{W}_{\text{xh}}$ do learn, so a wider embedding helps even if its rows stay close to random: with $d = 512$ instead of 64, $\mathbf{W}_{\text{xh}}$ has more random features of each token to combine. Dropout (Section 4.6), applied to the embeddings at rate 0.1 and to the hidden states before the output layer at rate 0.5, counters the overfitting.

8.5.3.3 A Wider Embedding with Dropout

This is where the two dropout rates of `RNNLMScratch` come into use. The new model has 740,355 parameters, most of them in the $1{,}027 \times 512$ embedding table; everything else in the training procedure stays the same.

```
rnn = RNNScratch(num_inputs=512, num_hiddens=128)
model = RNNLMScratch(rnn, vocab_size=len(data.vocab), lr=4, dropout=0.5,
                     emb_dropout=0.1)
```

```
trainer = d2l.Trainer(max_epochs=10, gradient_clip_val=1, num_gpus=1)
trainer.fit(model, data)
```



```
ppl_scratch = math.exp(trainer.history['val_loss'][-1])
print(f'validation perplexity {ppl_scratch:.1f}, '
      f'{math.log2(ppl_scratch) / bytes_per_token:.2f} bits per byte')
```

```
validation perplexity 59.9, 2.13 bits per byte
```

The validation perplexity falls to about 60, against about 90 to 100 for the plain model; in bits per byte, about 2.1 against 2.3 to 2.4. The training loss on the board is measured with dropout active, which raises it, so it no longer shows the gap between training and validation directly. The validation loss is still at or near its lowest after ten epochs.

8.5.3.4 Comparison with Kneser–Ney

Is this any good? The fair opponent is the best count-based model of Section 8.3, interpolated Kneser–Ney smoothing (Section 8.3.2.3). `d2l.kneser_ney` counts the n-grams of a training sequence and returns the natural log-probability of every token of another sequence, given the tokens before it. Both models must see the same text, so we count the n-grams of the RNN's 50,000 training tokens, in the same BPE tokenization. We score the validation tokens from the 33rd on, `val[32:]`, so that the RNN, below, can read a full window of 32 validation tokens before each of them.

```
first_tokens = d2l.numpy(data.X[:, 0]).tolist()  # First token of each window
train = first_tokens[:data.num_train]
val = first_tokens[data.num_train:data.num_train + data.num_val]
lp_kns = {n: d2l.kneser_ney(train, val, len(data.vocab), n)[32:]
          for n in (2, 3, 4)}
for n, lp in lp_kns.items():
    bits = -sum(lp) / len(lp) / math.log(2)
    print(f'KN {n}-gram: {bits / bytes_per_token:.2f} bits per byte')
lp_kn = lp_kns[3]
```

```
KN 2-gram: 2.24 bits per byte
KN 3-gram: 2.09 bits per byte
KN 4-gram: 2.08 bits per byte
```

The trigram is much better than the bigram, and a fourth token of context helps it little more. We now score the same tokens `val[32:]` with the RNN, each as the last target of a validation window, so that the RNN has read the 32 tokens before it: validation window j starts at `val[j]`, and its last target is `val[j + 32]`. This treats the RNN more kindly than the validation loss on the board, which averages over all 32 positions of a window, the first of which see only one or a few tokens of context. The cell computes the log-probability of every target of every window; we use the last column now and the others shortly.

```
start, stop = data.num_train, data.num_train + data.num_val - 32  # val[32:]
```

```
model.eval()
with torch.inference_mode():  # Log-probability of every target
    logp = d2l.numpy(torch.cat([
        F.log_softmax(model(X.to(d2l.try_gpu())), -1).gather(
```

```
            -1, Y.to(d2l.try_gpu())[..., None])[..., 0]
        for X, Y in zip(data.X[start:stop].split(1024),
                        data.Y[start:stop].split(1024))]))
```

Before we look at the result, note that the two models have very different strengths. The RNN sees 32 tokens of context but learns only from what it has seen in a few hundred SGD steps. The counts see two tokens but remember every trigram of the training text. It is therefore natural to combine them. Linear interpolation, $\lambda P_{\text{RNN}} + (1-\lambda) P_{\text{KN}}$, is how Mikolov et al. (2011) combined their RNN language models with Kneser–Ney models, choosing λ on held-out data. We fix $\lambda = 0.5$ instead of tuning it, close to the best weight for this pair. A weighted average of two probability distributions is a probability distribution, so the mixture is a language model too.

```
lp_rnn = list(logp[:, -1])  # The last target of each window: val[32:]
lam = 0.5
lp_mix = [math.log(lam * math.exp(a) + (1 - lam) * math.exp(b))
          for a, b in zip(lp_rnn, lp_kn)]
for name, lp in [('RNN', lp_rnn), ('KN trigram', lp_kn), ('mixture', lp_mix)]:
    bits = -sum(lp) / len(lp) / math.log(2)
    print(f'{name}: {bits / bytes_per_token:.2f} bits per byte')
better = np.mean(np.array(lp_rnn) > np.array(lp_kn))
print(f'the RNN is the more confident model on {better:.0%} of the tokens')
```

```
RNN: 2.12 bits per byte
KN trigram: 2.09 bits per byte
mixture: 2.03 bits per byte
the RNN is the more confident model on 50% of the tokens
```

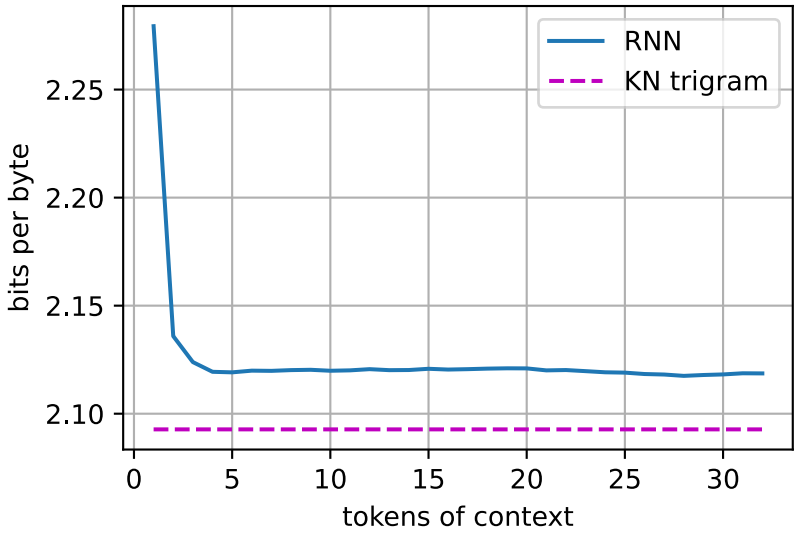
After all this effort, the RNN alone does not beat a trigram. It comes within a few hundredths of a bit per byte of Kneser–Ney, on the wrong side. (Both numbers are higher than those in the table of Section 8.3.2.4, which was computed on lowercased text with letters only. Bits per byte compares tokenizers on the same text, and on the raw text of this section the Kneser–Ney trigram needs about 2.1.) The mixture is the more interesting number: it is better than the trigram by about 0.06 bits per byte, which could not happen if the two models assigned the same probabilities to the same tokens. Indeed, each is the more confident of the two on about half of the tokens. The two models are about equally good, and they are wrong in different places.

Why does a model that reads 32 tokens do no better than one that reads two? Before we speculate, we can measure how much of its context the RNN uses. In every validation window the first target is predicted from one token of context, the second from two, and so on. Averaging the surprisal at each position over all validation windows gives the loss as a function of the context the model had, which we draw next to the Kneser–Ney trigram.

```
bits = -logp.mean(0) / math.log(2) / bytes_per_token  # One per position
kn_bits = -sum(lp_kn) / len(lp_kn) / math.log(2) / bytes_per_token
positions = np.arange(1, data.num_steps + 1)
d2l.plot(positions, [bits, [kn_bits] * data.num_steps],
         'tokens of context', 'bits per byte', legend=['RNN', 'KN trigram'],
         figsize=(4.5, 3))
```

```
print('bits per byte with 1, 2, 3, 4, 8, 32 tokens of context:',
      [round(float(bits[k - 1]), 2) for k in (1, 2, 3, 4, 8, 32)])
```

```
bits per byte with 1, 2, 3, 4, 8, 32 tokens of context: [2.28, 2.14, 2.12, 2.12, 2.12, 2.12]
```



The first target of a window, predicted from a single token, costs the most, and the second and third much less. From about the fourth position on the curve is flat: a model that has read thirty tokens predicts no better than one that has read three. This answers the question. The trained RNN uses only the last few tokens of its window, about as many as the trigram sees, and so the two tie. Its state could carry information across all 32 steps. Training did not teach it to.

One cause of the shortfall we have already met: SGD left the embedding table almost untrained. Adam (Section 9.5), an optimizer that adapts the step to each parameter, moves the rows much further. Trained with it and with slightly stronger embedding dropout, the RNN alone edges past the trigram; an exercise asks you to try it.

8.5.4 Text Generation

A trained language model can continue a prefix (Section 8.4). The `predict` method below first *warms up* on the prefix, feeding its tokens one at a time to build up a hidden state without emitting anything. It then generates `num_tokens` new tokens, each time turning the current logits **o** into a choice of next token and feeding that token back in as the next input.

By default the choice is *greedy*: the method takes the highest-scoring token at every step. With a positive `temperature` it samples from the softmax of the scaled logits instead, a rule we leave to Section 8.7.

```
@d2l.add_to_class(RNNLMScratch)
@torch.inference_mode()  # Generation needs no autograd graph
def predict(self, prefix, num_tokens, tok, device=None, temperature=0.0):
```

```
    outputs, state = tok.encode(prefix), None
    for i in range(len(outputs) - 1):  # Warm up on the prefix
        X = d2l.tensor([[outputs[i]]], device=device)
        _, state = self.rnn(self.embedding(X), state)
    for _ in range(num_tokens):  # Generate num_tokens continuation tokens
        X = d2l.tensor([[outputs[-1]]], device=device)
        rnn_outputs, state = self.rnn(self.embedding(X), state)
        logits = self.output_layer(rnn_outputs)[0, 0]
        if temperature == 0:  # Greedy: the highest-scoring token
            token = logits.argmax()
        else:  # Sample from the softmax of logits / temperature
            token = torch.distributions.Categorical(
                logits=logits / temperature).sample()
        outputs.append(int(token))
    return tok.decode(outputs)
```

We start with greedy decoding. The prefix is real text, not token ids: the tokenizer trained by the pipeline (`data.tokenizer`) encodes it on the way in and decodes the generated ids on the way out.

```
d2l.print_wrapped(repr(model.predict('The Time Traveller', 50,
                                     data.tokenizer, d2l.try_gpu())))
```

```
'The Time Traveller,\nthe\nI could see\nthe\nI could see\nthe\nI could
    see\nthe\nI could see\nthe\nI could see\nthe\nI could see\nthe\nI could
    see\nthe\nI could see\n'
```

The continuation has the look of English for a few words, and then it often gets stuck. Greedy decoding is deterministic, and it tends to fall into a loop: once the state returns to a phrase it has already produced, the phrase comes back again and again. How to choose tokens better is the subject of Section 8.7.

8.5.5 Concise Implementation

Frameworks ship recurrent layers that bundle the cell and the loop over time steps into one optimized call. The `RNN` class below wraps the framework layer and keeps the interface of `RNNScratch`: it consumes a (`num_steps`, `batch_size`, `num_inputs`) tensor and an optional state, and returns the hidden states of all steps together with the final state. So that the two implementations train the same model, `RNN` also initializes the layer as `RNNScratch` does, with weights from $\mathcal{N}(0, \sigma^2)$ and zero biases.

`nn.RNN` keeps two bias vectors, one for the input term and one for the recurrent term. Their sum plays the role of $\mathbf{b}_\text{h}$, and both start at zero.

```
class RNN(d2l.Module):
    """The RNN model implemented with high-level APIs."""
    def __init__(self, num_inputs, num_hiddens, sigma=0.01):
        super().__init__()
        self.save_hyperparameters()
        self.rnn = nn.RNN(num_inputs, num_hiddens)
        for name, param in self.rnn.named_parameters():
```

```
            if name.startswith('weight'):
                nn.init.normal_(param, std=sigma)
            else:  # Both bias vectors start at zero
                nn.init.zeros_(param)

    def forward(self, inputs, H=None):
        return self.rnn(inputs, H)
```

Because the interface is unchanged, the language model carries over by inheritance: `RNNLM` subclasses `RNNLMScratch` and swaps the hand-rolled embedding table and output layer for the framework's embedding and dense layers. Training, evaluation, and `predict` are all inherited. The new layers are initialized like their scratch counterparts, with unit-normal embedding rows, output weights from $\mathcal{N}(0, \sigma^2)$, and a zero output bias.

`nn.Embedding` draws its rows from $\mathcal{N}(0, 1)$ by default; the weights of `nn.Linear` are set explicitly.

```
class RNNLM(d2l.RNNLMScratch):
    """The RNN-based language model implemented with high-level APIs."""
    def init_params(self):
        self.emb = nn.Embedding(self.vocab_size, self.rnn.num_inputs)
        self.linear = nn.Linear(self.rnn.num_hiddens, self.vocab_size)
        nn.init.normal_(self.linear.weight, std=self.sigma)
        nn.init.zeros_(self.linear.bias)

    def embedding(self, X):
        return self.emb(X.T)

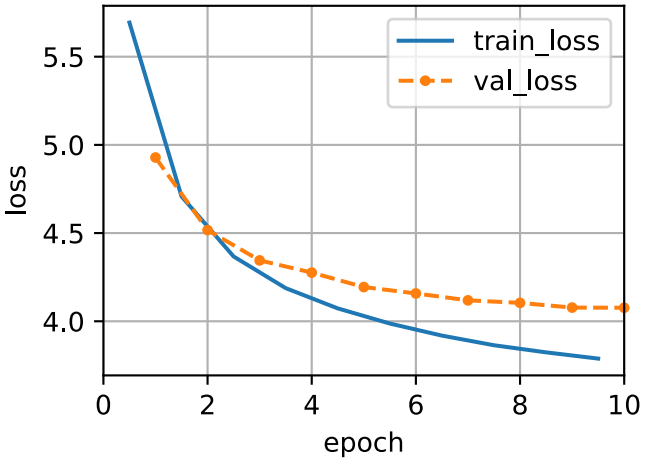
    def output_layer(self, hiddens):
        return d2l.swapaxes(self.linear(hiddens), 0, 1)
```

We build the concise model with the settings of the wider scratch model.

```
rnn = RNN(num_inputs=512, num_hiddens=128)
model = RNNLM(rnn, vocab_size=len(data.vocab), lr=4, dropout=0.5,
              emb_dropout=0.1)
```

Training is the same call as before, with the same hyperparameters.

```
trainer = d2l.Trainer(max_epochs=10, gradient_clip_val=1, num_gpus=1)
trainer.fit(model, data)
```



With the same initialization, data, and training, the concise model reaches about the same validation perplexity as the scratch model, and its greedy continuation gets stuck in the same way.

```
ppl_concise = math.exp(trainer.history['val_loss'][-1])
pred = model.predict('The Time Traveller', 30, data.tokenizer, d2l.try_gpu())
d2l.print_wrapped(f'perplexity {ppl_concise:.1f}, {pred!r}')
```

```
perplexity 59.0, 'The Time Traveller, and I had a fright of the Time
    Traveller. I had a faintness. I had a faintness. I had a'
```

The framework layer runs the same recurrence in a single optimized call, so the rest of the book writes recurrent cells from scratch only to explain them, and trains with the framework layers.

A tie with a trigram is less than the hidden state seemed to promise. The loss by position gave the immediate reason: the trained model uses only the last few tokens of its context. The deeper reason must lie in training. To learn that a token thirty steps back matters, the model needs a gradient that reaches thirty steps back, and that gradient passes through the same transition at every step. Section 8.6 computes it, and then measures it in this very model.

8.5.6 Summary

An RNN language model applies the same step at every position: it consumes a token and the previous state and returns logits and a new state of the same shape. The step is an embedding lookup, a tanh cell, and an output layer, and each framework's recurrent layer computes the same model as our tensor code. Training relies on two devices. Clipping the gradient norm bounds the change any one update makes, and without it training at our learning rate fails. Starting each window from a zero state truncates backpropagation at the window length. On *The Time Machine*, with a wide embedding and dropout, the RNN needs about 2.1 bits per byte, within a few hundredths of a Kneser–Ney trigram on the same text, and the average of the two models beats both. The loss by position shows why the RNN does no better: it uses only the last few tokens of its 32-token window. Greedy generation tends to fall into loops.

8.5.7 Exercises

1. code **Gradient clipping ablation.** `d2l.Trainer` applies the projection Equation 8.19 whenever `gradient_clip_val` is positive.
    1. Record the norm of the gradient before clipping at every update of the first epoch, and plot it. How often does clipping act, and when?
    2. The unclipped run of this section failed at `lr=4`. Halve the learning rate until three unclipped runs in a row train without a jump in the validation perplexity. How does their validation perplexity after ten epochs compare with that of the clipped run at `lr=4`? What, then, does clipping make possible?
    3. Replace tanh by ReLU in `RNNScratch` and train again, with and without clipping. Is clipping still needed? Explain the difference from the tanh cell in terms of the range of the hidden state.
2. code **Usable history.** Every training window starts from a zero state, so the model is never trained to use more than `num_steps` tokens of context, and the loss by position showed that it uses far fewer. `predict`, by contrast, carries the state across the whole prefix and every generated token. Evaluate the trained model on 512 consecutive validation tokens in two ways: resetting the state to zero every 32 tokens as in training, and carrying the state across all 512 tokens. Compare the average cross-entropy of the two. Does the carried state help, hurt, or make no difference, and what does the answer say about the states the model reaches beyond its training horizon?
3. code **Hyperparameters and data budget.**
    1. Starting from the model with the wider embedding, adjust the number of epochs, hidden units, embedding dimension, `num_steps`, learning rate, and dropout rates, or add weight decay, to lower the validation perplexity. Can the RNN alone, trained with SGD, beat the Kneser–Ney trigram? How does the relative improvement in bits per byte compare with the relative improvement in perplexity, and why?
    2. Set `num_train=10000` and train for 100 epochs. Compare the training and validation perplexity curves and explain what you see. Consecutive windows overlap by 31 tokens, so how many parameters does the model have per distinct training token in this regime?
4. code **Adam.** Train the model with the wider embedding with the Adam optimizer of Section 9.5 instead of SGD: subclass `RNNLMScratch`, override `configure_optimizers` to return Adam with learning rate 0.003, and keep clipping at 1.
    1. With `emb_dropout=0.2`, score the tokens `val[32:]` as in the comparison above, and compare its bits per byte with the SGD model, the Kneser–Ney trigram and 4-gram, and the mixture of the RNN and the trigram at $\lambda = 0.5$.
    2. Record the validation loss after every epoch, with `emb_dropout=0.1` and with 0.2. At which epoch is it lowest, and what happens after that?
    3. Measure again how far the embedding rows move during training. Why does Adam move them further than SGD does?
5. code **Vocabulary size and bits per byte.** ⍰ Retrain the model with the wider embedding for `vocab_size` 512, 1,024, 2,048, and 4,096 in `d2l.TimeMachine`. The number of windows depends on the vocabulary size, so split by fraction: build each dataset once to read off `n = len(data.X)`, then build it again with `num_train=int(0.8 * n)` and `num_val=n - int(0.8 * n)`. (Setting `data.num_train` afterwards would skip the step that keeps val-

idation tokens out of the training targets.) Every model then trains and validates on about the same stretch of text. Their per-token perplexities are not comparable, so compare them by bits per byte.

   1. Which vocabulary size gives the lowest bits per byte?
   2. How do the parameter counts of the embedding table and the output layer, and the length of the corpus in tokens, change with the vocabulary size? Which of these effects best explains the trend you measured?

6. code **Weight tying.** ⊠ Weight tying sets the embedding dimension equal to the number of hidden units and uses $\mathbf{E}^\top$ as the output projection in place of $\mathbf{W}_{\text{hq}}$ (Press and Wolf 2017; Inan et al. 2017).

   1. How many parameters does tying save at $|\mathcal{V}| = 1{,}027$ with 128 hidden units? GPT-2 small ties its $50{,}257 \times 768$ token embedding to its output layer. How many parameters would untying add there, and what fraction of the model's roughly 124 million parameters is that?
   2. Tie the weights in `RNNLMScratch` with an embedding dimension of 128. `RNNLMScratch` initializes $\mathbf{E}$ at unit scale and $\mathbf{W}_{\text{hq}}$ at scale `sigma`. What happens to the initial perplexity once the two are tied, and how do you restore a near-uniform initial prediction?
   3. Train the tied model and an untied model with a 128-dimensional embedding and 128 hidden units, and compare their validation perplexities.

7. code **Cross-book generalization.** ⊠ Subclass `d2l.TimeMachine` and override `_download` to return the text of a different H. G. Wells novel, such as *The War of the Worlds* (Project Gutenberg ebook 36). Train the tokenizer and the model on it.

   1. Evaluate the model on held-out text of its own novel and on *The Time Machine*, encoding both with the new novel's tokenizer. Report perplexity and bits per byte for both.
   2. Write $s = \log_2(\text{ppl})$ for the per-token surprisal and b for the bytes per token of an evaluated text, so that bits per byte is s/b. The ratio of bits per byte on *The Time Machine* (TM) to bits per byte on the held-out text of the new novel (own) factors as
   $$\frac{s_{\text{TM}}}{s_{\text{own}}} \cdot \frac{b_{\text{own}}}{b_{\text{TM}}}.$$
   The first factor reflects the model's predictions and the second the tokenizer's compression. Measure both factors. Which lies further from one?

8.6 Backpropagation Through Time

The RNN language model of Section 8.5 left us with two observations. Without gradient clipping, its training fell apart. With clipping it trained, but it learned to use no more than the last few tokens of its 32-token window. Both have the same origin. A recurrent network applies the same weights at every time step, so a gradient that travels backward through the sequence is multiplied by a similar factor at each step. Over many steps these factors compound, the way interest compounds a deposit, and the gradient grows or shrinks geometrically.

The simplest recurrence shows the effect. Consider a scalar state that keeps a fraction w_{h} of its value at each step and adds the new input, $h_t = w_{\text{h}} h_{t-1} + x_t$. Unrolling it gives

$h_t = \sum_{s \le t} w_{\mathrm{h}}^{t-s} x_s$, so the derivative of h_t with respect to the state k steps earlier is w_{h}^k. With $w_{\mathrm{h}} = 0.9$, forty steps shrink this derivative to $0.9^{40} \approx 0.015$. With $w_{\mathrm{h}} = 1.1$, they inflate it to $1.1^{40} \approx 45$, as forty years at 10% interest multiply a deposit 45-fold. Over long sequences the gradient either fades or blows up.

Both outcomes hurt. A gradient that explodes destabilizes training, which is why we had to clip. A gradient that vanishes is harder to notice and does more damage. The model receives almost no signal about how a token forty steps back should have changed its prediction now, because that signal is buried under the gradients of nearby tokens, so it learns the dependency slowly if at all.

To compute the gradient of a recurrent network, we unroll the recurrence over time into a feedforward network whose layers share their weights, and apply the backpropagation of Section 4.3 to it. This is called *backpropagation through time* (Werbos 1990). As an algorithm it is nothing new; what is new is the shape of the result. The gradient turns out to be a sum over earlier steps, and each term carries a product of Jacobians. For a linear recurrence this product is a matrix power, whose eigenvalues decide whether the gradient vanishes or explodes. We then measure how far back the gradient reaches in the model we trained. Finally, we bound the cost of backpropagating through long sequences by truncation and ask what truncation gives up.

8.6.1 The Unrolled Computational Graph

Let us start with scalars. They show the whole dependency structure, and the vector case merely replaces each derivative by a Jacobian. Write h_t, x_t, and $o_t \in \mathbb{R}$ for the state, the input, and the output at step t. The state update g has parameter w_{h} and the output map f has parameter w_{o}, the same at every step:

$$\begin{aligned} h_t &= g(h_{t-1}, x_t, w_{\mathrm{h}}), \\ o_t &= f(h_t, w_{\mathrm{o}}). \end{aligned} \tag{8.20}$$

The objective averages the per-step loss over all T steps, $L = \frac{1}{T} \sum_{t=1}^{T} l(y_t, o_t)$.

The unrolled network has one layer per time step, and every layer uses the same w_{h}. Backpropagation through it is ordinary backpropagation with one rule for shared parameters: the gradient of a parameter that is used in many places is the sum of its gradients at each place, just as the gradient of a convolution kernel sums over the positions where the kernel is applied.

The output parameter w_{o} is easy, since it reaches L only through each o_t directly. The state parameter w_{h} is harder: it affects every state, and every state affects all later ones. By the chain rule,

$$\frac{\partial L}{\partial w_{\mathrm{h}}} = \frac{1}{T} \sum_{t=1}^{T} \frac{\partial l(y_t, o_t)}{\partial o_t} \frac{\partial f(h_t, w_{\mathrm{o}})}{\partial h_t} \frac{\partial h_t}{\partial w_{\mathrm{h}}}. \tag{8.21}$$

The first two factors are local to step t. The third is not, because h_t depends on w_{h} directly and also through h_{t-1}, which depends on w_{h} in turn:

$$\frac{\partial h_t}{\partial w_\mathrm{h}} = \frac{\partial g(h_{t-1}, x_t, w_\mathrm{h})}{\partial w_\mathrm{h}} + \frac{\partial g(h_{t-1}, x_t, w_\mathrm{h})}{\partial h_{t-1}} \frac{\partial h_{t-1}}{\partial w_\mathrm{h}}. \tag{8.22}$$

This recursion has the form $a_t = u_t + c_t a_{t-1}$ with $a_0 = 0$, since the initial state does not depend on w_h. Substituting the recursion into itself $t-1$ times gives, for $t \geq 1$,

$$a_t = u_t + \sum_{i=1}^{t-1} \Big(\prod_{j=i+1}^{t} c_j \Big) u_i. \tag{8.23}$$

Each u_i enters once, multiplied by all the c_j that come after it. With $a_t = \partial h_t / \partial w_\mathrm{h}$, $u_t = \partial g / \partial w_\mathrm{h}$, and $c_t = \partial g / \partial h_{t-1}$, the recursion Equation 8.22 unrolls into

$$\frac{\partial h_t}{\partial w_\mathrm{h}} = \frac{\partial g(h_{t-1}, x_t, w_\mathrm{h})}{\partial w_\mathrm{h}} + \sum_{i=1}^{t-1} \Big(\prod_{j=i+1}^{t} \frac{\partial g(h_{j-1}, x_j, w_\mathrm{h})}{\partial h_{j-1}} \Big) \frac{\partial g(h_{i-1}, x_i, w_\mathrm{h})}{\partial w_\mathrm{h}}. \tag{8.24}$$

The term for step i is the effect of w_h at that step, carried forward to step t by the product of the $t-i$ state Jacobians $\partial g / \partial h_{j-1}$ in between. For the recurrence of the opening, $g = w_\mathrm{h} h_{t-1} + x_t$, every one of these Jacobians equals w_h, and the term from $t-i$ steps back is scaled by w_h^{t-i}. The full sum is exact, but its cost grows with t, and its long products are where the numbers go wrong. To see how, we need the vector case.

8.6.2 Vanishing and Exploding Gradients

In the vector case the product of Jacobians becomes a product of matrices. Take the RNN of Section 8.4 and drop the biases and the nonlinearity, a crude simplification to which we return below. For a minibatch of b examples with inputs $\mathbf{X}_t \in \mathbb{R}^{b \times d}$, the states and outputs are

$$\mathbf{H}_t = \mathbf{X}_t \mathbf{W}_\mathrm{xh} + \mathbf{H}_{t-1} \mathbf{W}_\mathrm{hh}, \qquad \mathbf{O}_t = \mathbf{H}_t \mathbf{W}_\mathrm{hq}, \tag{8.25}$$

with $\mathbf{W}_\mathrm{xh} \in \mathbb{R}^{d \times h}$, $\mathbf{W}_\mathrm{hh} \in \mathbb{R}^{h \times h}$, and $\mathbf{W}_\mathrm{hq} \in \mathbb{R}^{h \times q}$, as in Section 8.4. Figure 8.8 draws three steps of this network, with the forward pass in grey and the backward pass in red.

The scalar derivation followed the influence of the weight forward from each step. Backpropagation computes the same sum from the other end: it carries the gradient of the loss backward, one state at a time. Write $\mathbf{G}_t = \partial L / \partial \mathbf{H}_t$ for the gradient of the objective with respect to the state, a matrix of the same shape as $\mathbf{H}_t$. The state $\mathbf{H}_t$ affects L along two paths, through its output $\mathbf{O}_t$ and through the next state $\mathbf{H}_{t+1}$. Backpropagating through a matrix product multiplies the gradient by the transpose of the weight (Section 4.3), so

$$\mathbf{G}_t = \frac{\partial L}{\partial \mathbf{O}_t} \mathbf{W}_\mathrm{hq}^\top + \mathbf{G}_{t+1} \mathbf{W}_\mathrm{hh}^\top, \qquad \mathbf{G}_{T+1} = \mathbf{0}. \tag{8.26}$$

These are the two red arrows into each state of Figure 8.8. The recursion runs backward in time, from $\mathbf{G}_T$ to $\mathbf{G}_1$. Otherwise it has the form of Equation 8.23 with matrices, and it unrolls in the same way:

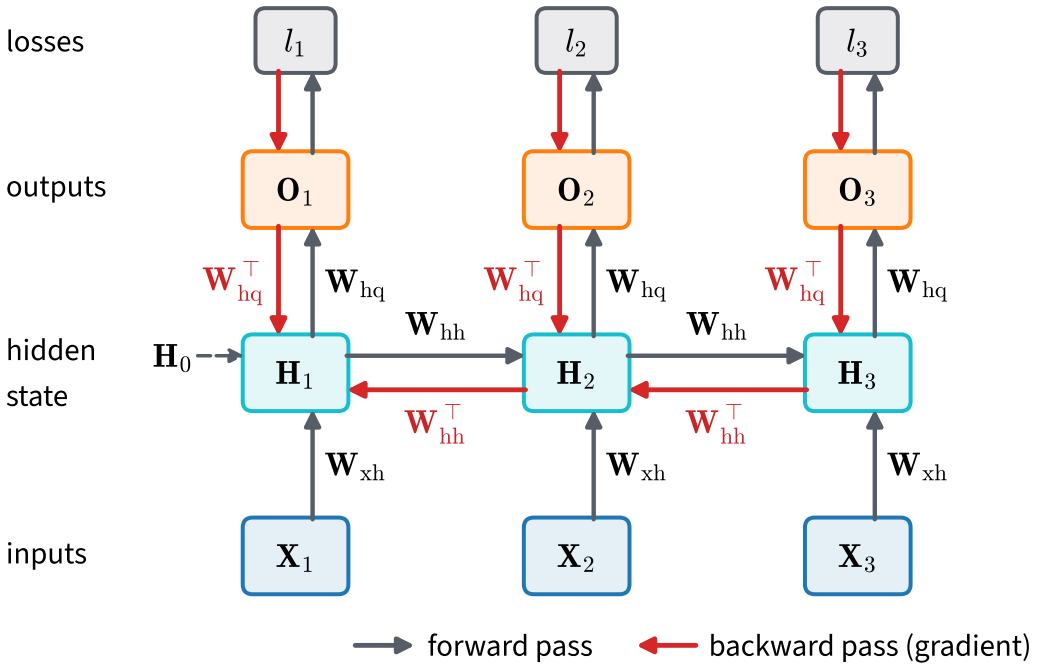


Figure 8.8: Forward pass (grey) and backward pass (red) of a linear RNN unrolled over three time steps. The gradient that reaches each hidden state except the last is the sum of two red arrows: one from its own output, through $\mathbf{W}_{\text{hq}}^\top$, and one from the next state, through $\mathbf{W}_{\text{hh}}^\top$.

$$\mathbf{G}_t = \sum_{j=t}^{T} \frac{\partial L}{\partial \mathbf{O}_j} \mathbf{W}_{\text{hq}}^\top \left(\mathbf{W}_{\text{hh}}^\top\right)^{j-t}. \tag{8.27}$$

The loss at step j reaches the state $k = j - t$ steps earlier through the k^{th} power of $\mathbf{W}_{\text{hh}}^\top$. The parameter gradients add one term per step, each containing $\mathbf{G}_t$:

$$\frac{\partial L}{\partial \mathbf{W}_{\text{hh}}} = \sum_{t=1}^{T} \mathbf{H}_{t-1}^\top \mathbf{G}_t, \qquad \frac{\partial L}{\partial \mathbf{W}_{\text{xh}}} = \sum_{t=1}^{T} \mathbf{X}_t^\top \mathbf{G}_t. \tag{8.28}$$

Whatever happens to $\mathbf{G}_t$ therefore happens to the gradients of the weights as well.

How large is the k^{th} power of a matrix? We have met this question twice, in Section 1.3.5 and for the layers of a deep network in Section 4.4: the power multiplies each eigenvector by λ^k, and the largest $|\lambda|$ soon dominates. $\mathbf{W}_{\text{hh}}^\top$ has the same eigenvalues as $\mathbf{W}_{\text{hh}}$; let ρ be the largest of their magnitudes, the *spectral radius*. For large k the norm of the power behaves like ρ^k. If $\rho < 1$, the contribution from far ahead vanishes; if $\rho > 1$, it explodes. This is the scalar w_{h}^k of the opening, with ρ in the role of $|w_{\text{h}}|$. Unlike the layers of a deep network, the steps of a recurrence all contribute the same matrix. At $\rho = 0.9$ and $\rho = 1.1$, forty steps scale a contribution by about 0.015 and 45, the numbers of the opening. A contribution shrunk to 1.5% is lost among the nearer ones; one amplified 45-fold swamps them.

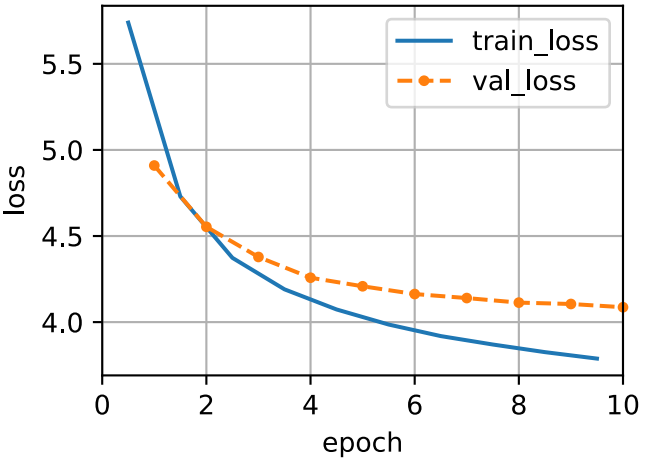
The model of Section 8.5 is not this tidy: it has a tanh, and its $\mathbf{W}_{\text{hh}}$ is a general, nonsymmetric matrix. Does the analysis survive? In outline, yes. The powers of a nonsymmetric matrix can grow for a while even when $\rho < 1$, but the ρ^k trend wins for large k. The tanh multiplies the gradient of each unit by its derivative $1 - h^2$, a number between 0 and 1, so it never lengthens a gradient that passes through it. It pushes toward vanishing, most strongly when units saturate, and it does not rule out explosion when $\mathbf{W}_{\text{hh}}$ is large and the states are near zero.

So much for what can happen. What does happen in the model we trained? To find out, we train the RNN language model of Section 8.5 again, with the wider embedding and dropout, for ten epochs.

```
data = d2l.TimeMachine(batch_size=1024, num_steps=32)
```

```
rnn = d2l.RNNScratch(num_inputs=512, num_hiddens=128)
model = d2l.RNNLMScratch(rnn, vocab_size=len(data.vocab), lr=4, dropout=0.5,
                         emb_dropout=0.1)
```

```
trainer = d2l.Trainer(max_epochs=10, gradient_clip_val=1, num_gpus=1)
trainer.fit(model, data)
```



We feed the trained model 256 stretches of 64 validation tokens, twice the window it was trained on, and keep its states and the logits of its last step.

```
L = 64
val = d2l.numpy(data.X[data.num_train:data.num_train + data.num_val, 0])
starts = np.random.default_rng().choice(len(val) - L, 256, replace=False)
seqs = np.stack([val[s:s + L] for s in starts])  # (256, L) token ids
targets = val[starts + L]                        # The token after each stretch
```

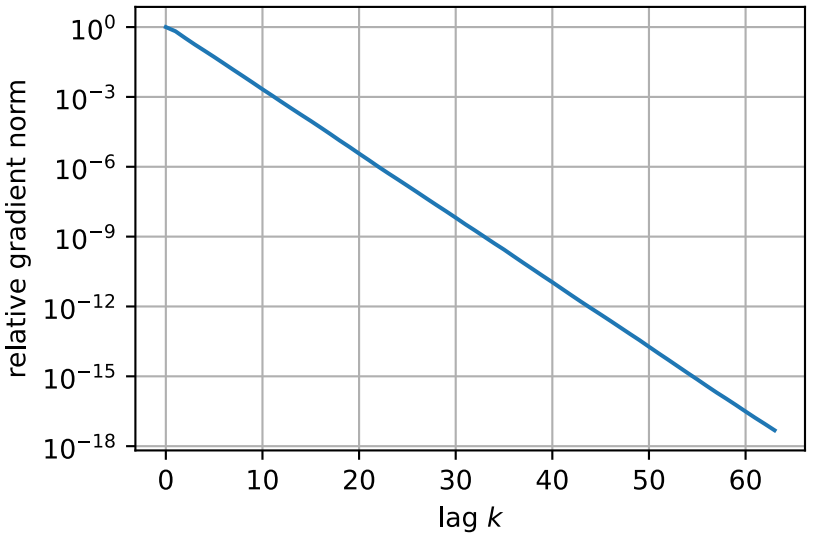
```
with torch.inference_mode():
    states, _ = model.rnn(model.embedding(
        d2l.tensor(seqs, device=d2l.try_gpu())))
    logits = model.output_layer(states[-1:])[:, 0]  # The last step only
H, logits = d2l.numpy(d2l.stack(states, 0)), d2l.numpy(logits)
W_hh, W_hq = d2l.numpy(model.rnn.W_hh), d2l.numpy(model.W_hq)
```

We backpropagate the loss at the last step by hand. Its gradient with respect to the logits is the softmax minus the one-hot target (Section 3.1.2.2), and $\mathbf{W}_{\text{hq}}^\top$ carries it to the last state. From there, each step back multiplies the gradient entrywise by the tanh derivative $1 - \mathbf{H}_t^2$ and then by $\mathbf{W}_{\text{hh}}^\top$: this is Equation 8.26 with the tanh, and with no output term, since we follow a single loss. We record the norm of the gradient at every lag k, relative to its norm at the last state, and average over the 256 stretches on a logarithmic scale.

```
P = np.exp(logits - logits.max(1, keepdims=True))
P /= P.sum(1, keepdims=True)
P[np.arange(len(targets)), targets] -= 1  # dL/dO: softmax minus one-hot
G = (P @ W_hq.T).astype(np.float64)       # dL/dH at the last step
norms = []
for t in range(L - 1, -1, -1):            # Backward through time
    norms.append(np.linalg.norm(G, axis=1))
    G = (G * (1 - H[t] ** 2)) @ W_hh.T    # Through the tanh, then W_hh
rel = np.exp(np.log(np.array(norms) / norms[0]).mean(1))
print(f'spectral radius of W_hh: {np.abs(np.linalg.eigvals(W_hh)).max():.2f}; '
```

```
      f'mean tanh derivative: {(1 - H ** 2).mean():.2f}')
print('relative gradient norm at lags 1, 2, 4, 8, 16, 32:',
      [float(f'{rel[k]:.2g}') for k in (1, 2, 4, 8, 16, 32)])
d2l.plot(np.arange(L), rel, 'lag $k$', 'relative gradient norm',
         yscale='log', figsize=(4.5, 3))
```

```
spectral radius of W_hh: 1.64; mean tanh derivative: 0.27
relative gradient norm at lags 1, 2, 4, 8, 16, 32: [0.66, 0.34, 0.097, 0.0077, 4.9e-05, 1.8e-09]
```



The result is not what the linear analysis leads us to expect. The spectral radius of the trained $\mathbf{W}_{\text{hh}}$ is well above 1, near 1.8, and a linear recurrence with this matrix would have exploding gradients. Yet the gradient vanishes. It loses about a third of its size at the first step back, falls below 1% within about eight steps, and at the start of a 32-token window it is a few billionths of what it was. The tanh makes the difference. Its derivative averages about 0.27 over the states, which means that many units sit close to saturation and pass back little gradient. As a rough estimate, a factor near 1.8 from the weights and 0.27 from the tanh leave about half of the gradient at each step, and the measured line indeed falls by a little less than half per step.

The memory of this model is thus limited by its gradients long before the window cuts them off. We have seen the same limit from the other side: the loss by position in Section 8.5 stopped improving after the third token of context. A model trained by gradient descent cannot learn to use context from which no gradient reaches it.

8.6.2.1 Remedies for Exploding and Vanishing Gradients

The two failure modes need different remedies, and explosion is the easier one. Its immediate danger is a single update so large that it wrecks the parameters. Gradient clipping Equation 8.19 bounds the step: a gradient longer than θ is scaled down to norm θ before the update, so no spike can move the parameters far. Note, however, that clipping does not repair the gradient itself. The gradient is a sum over lags, as in Equation 8.27, and rescaling multiplies every term by the same factor, so the balance between near and distant lags is

unchanged. For a vanishing gradient that balance is the whole problem. The terms from distant steps are tiny next to the recent ones, and there is no large step to limit.

The remedy for vanishing gradients has to change the recurrence itself, so that the product of Jacobians does not shrink along the directions the model needs for memory. Two designs do this. *Gated* cells such as the LSTM add a path along which the state is copied from step to step, scaled by a gate that the network sets close to 1 when it needs to remember; along this path each Jacobian is close to the identity. *Linear recurrences* drop the nonlinearity from the state update and parameterize the state matrix so that its eigenvalues have magnitude at most 1 by construction. Their gradients cannot explode, and the network can learn eigenvalues close to 1 for the directions that must remember. Both designs are the subject of Chapter 12.

8.6.3 Truncated Backpropagation Through Time

Full backpropagation through time has a more mundane problem as well, its cost. The backward recursion Equation 8.26 needs every state $\mathbf{H}_1, \dots, \mathbf{H}_T$ of the forward pass, so memory grows linearly with the sequence length, and every parameter update requires a forward and a backward pass over the whole sequence. *The Time Machine* is about 66,000 BPE tokens long; treated as a single sequence, it would allow one update per pass through the book, and the backward pass would have to store all 66,000 states.

Truncation bounds both costs. We cut the stream into segments of a fixed length and backpropagate only within each segment, so that the sum Equation 8.27 stops at the end of the segment. Figure 8.9 contrasts the two.

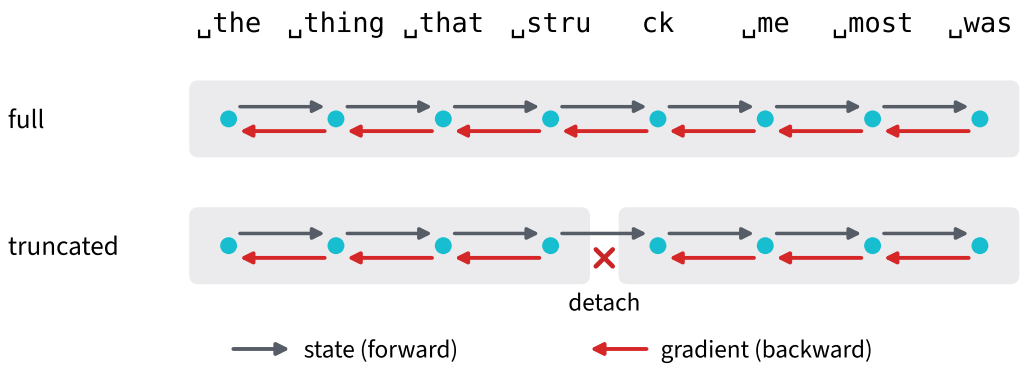


Figure 8.9: Full and truncated backpropagation through time over a token sequence. Both run the same forward pass (grey). Full backpropagation sends the gradient (red) back along the entire sequence. Truncation cuts the sequence into segments of four tokens and detaches the state at each boundary: its value crosses into the next segment, but no gradient flows back across.

Truncation drops every term of Equation 8.27 that crosses a segment boundary, so the truncated gradient is biased, and biased toward short dependencies (Jaeger 2002). How much is dropped depends on where a loss sits in its segment. In a segment of 32 tokens, the loss at the last position reaches back 31 steps and the loss at the first reaches no earlier state at all, about 16 steps on average. This is a hard limit. No gradient ever connects two tokens

in different segments, so the model cannot be trained to use a dependency longer than a segment, however well its recurrence preserves the state. The windows of 32 tokens in Section 8.5 are such segments in their simplest form: every window starts from a zero state, so neither state nor gradient crosses a boundary.

Figure 8.9 shows the variant that is common for long sequences. The final state of one segment becomes the initial state of the next, so the forward pass sees the whole history. The carried state is *detached* from the computational graph: its value crosses the boundary, but backpropagation stops there, and the cost of a backward pass does not grow with every segment. The gradient is cut all the same. The model may come to rely on an old state, but it is never told how that state should have been different.

Simple as it is, truncation at a fixed length remains the common choice. For a plain recurrence it also costs less than the hard limit suggests: unless the spectral radius sits close to 1, the signal from a few dozen steps back has faded geometrically before truncation removes it. For our trained model, as we measured, the fading sets the limit, well inside the window. By this measurement a longer window alone would change little (the last exercise tests the prediction). What the model lacks is a recurrence that preserves its gradient, and no change to the training procedure can supply one.

8.6.4 Summary

Backpropagation through time is backpropagation on the unrolled recurrence: the network becomes a feedforward network with one layer per time step, and each shared weight sums its gradient over all steps. The gradient that reaches back k steps carries a product of k state Jacobians, which for a linear RNN is the k^{th} power of $\mathbf{W}_{\text{hh}}^\top$ and scales like ρ^k. A spectral radius below 1 makes distant contributions vanish; one above 1 makes them explode. A tanh shortens the gradient further at every step: in our trained model the spectral radius is near 1.8, and still the gradient falls below 1% within about eight steps. Clipping controls explosion, while vanishing requires a change of architecture, such as gating or a linear recurrence with stable eigenvalues. Truncation bounds the memory and the compute of the backward pass, but a model trained with it cannot learn dependencies longer than a segment.

8.6.5 Exercises

1. **Backward pass by hand.** One step of `RNNScratch` (Section 8.5) computes

$$\mathbf{H}_t = \tanh(\mathbf{X}_t\mathbf{W}_{\text{xh}} + \mathbf{H}_{t-1}\mathbf{W}_{\text{hh}} + \mathbf{b}_{\text{h}}).$$

   Let $\mathbf{G}_t = \partial L/\partial \mathbf{H}_t$ be the gradient arriving from later steps and from the output at step t.

   1. Derive the contributions of this step to $\partial L/\partial \mathbf{W}_{\text{xh}}$, $\partial L/\partial \mathbf{W}_{\text{hh}}$, $\partial L/\partial \mathbf{b}_{\text{h}}$, and $\partial L/\partial \mathbf{H}_{t-1}$ as functions of $\mathbf{G}_t$, $\mathbf{X}_t$, $\mathbf{H}_t$, $\mathbf{H}_{t-1}$, and $\mathbf{W}_{\text{hh}}$.
   2. Adding the output contribution at step $t-1$ to the contribution to $\partial L/\partial \mathbf{H}_{t-1}$ gives a recursion for $\mathbf{G}_{t-1}$. Show that it reduces to Equation 8.26 when the tanh and the bias are dropped, and name the factor that the tanh adds at every step. Explain also how the parameter gradients accumulate over the T steps of a window, and compare with Equation 8.28.

   Adapted from CMU 11-785, Homework 3 Part 1.

2. **Power iteration and alignment.** Let $\mathbf{M} \in \mathbb{R}^{h\times h}$ be symmetric with eigenvalues λ_i and eigenvectors $\mathbf{v}_i$ for $i = 1, \dots, h$, ordered so that $|\lambda_i| \geq |\lambda_{i+1}|$.
    1. Show that $\mathbf{M}^k$ has eigenvalues λ_i^k with the same eigenvectors.
    2. Show that for a random vector $\mathbf{x} \in \mathbb{R}^h$ with independent standard normal entries, $\mathbf{M}^k\mathbf{x}$ becomes aligned with $\mathbf{v}_1$ as k grows. State the condition on the eigenvalues that this requires and the sense in which the alignment holds.
    3. What does this imply for the gradient Equation 8.27 of an RNN, beyond the change in its norm?
3. **Beyond clipping.** Propose two ways of coping with exploding gradients in a plain RNN other than clipping, truncation, and the two architectural remedies of Section 8.6.2.1: one that acts on the parameters and one that acts on the optimizer step. For each, say whether it changes the product of Jacobians in Equation 8.27 or only limits what a large gradient does to the update, and whether it also helps against vanishing gradients.
4. code **Backward pass in code.** ⊠ Train `d2l.RNNLMScratch` on `d2l.TimeMachine` as in Section 8.5. Using the single-step gradients and the recursion of problem 1, implement the backward pass of its `RNNScratch` cell over a full window with tensor operations only, starting from the gradient of the loss with respect to the logits (Section 3.1.2.2). Run it on one training minibatch, and compare your gradients of `W_xh`, `W_hh`, and `b_h` with those computed by automatic differentiation.

    Adapted from CMU 11-785, Homework 3 Part 1.
5. code **Nonlinear Jacobians.** ⊠ Draw a random symmetric 100×100 matrix, rescale it to spectral radius $\rho \in \{0.9, 1.0, 1.1\}$, and plot the norm of its k^{th} power $\|\mathbf{J}^k\|$ against k on a logarithmic axis: three straight lines, as the eigenvalues predict. Then leave the symmetric linear case.
    1. Replace the symmetric matrix by a general, nonsymmetric $\mathbf{W}_{\text{hh}}$ rescaled to each radius, and plot $\|\mathbf{J}^k\|$ against k. Report the largest value of $\|\mathbf{J}^k\|$ at $\rho = 0.9$ and the lag at which it occurs. Explain why the ratio $\|\mathbf{J}^k\|/\rho^k$ can exceed one and why it stays bounded as k grows (a random matrix is diagonalizable with probability one).
    2. Return to the symmetric $\mathbf{W}_{\text{hh}}$ and run the tanh RNN
    $$\mathbf{H}_t = \tanh(\mathbf{H}_{t-1}\mathbf{W}_{\text{hh}} + \mathbf{X}_t), \qquad \mathbf{H}_0 = \mathbf{0}, \tag{8.29}$$
    for a single sequence ($\mathbf{H}_t \in \mathbb{R}^{1\times h}$) with inputs $\mathbf{X}_t$ whose entries are drawn independently from $\mathcal{N}(0, s^2)$. Plot the norm of the product of the Jacobians $\partial\mathbf{H}_j/\partial\mathbf{H}_{j-1}$ along the trajectory for $s = 0.1$ and $s = 1$. How do the nonlinearity and the input scale change the growth of the norm, and why?
    3. Explain why a largest singular value of $\mathbf{W}_{\text{hh}}$ below one is sufficient for the product to shrink, whereas a largest singular value above one is necessary but not sufficient for it to grow (Bengio et al. 1994; Pascanu et al. 2013). Relate the argument to your tanh curves at $\rho = 0.9$ and $\rho = 1.1$.
6. code **Window length and memory horizon.** ⊠ In the model of Section 8.6.2, the gradient of the last loss falls below 1% of its size within about eight steps, well inside the training window of 32 tokens.
    1. Before running anything: if the model is retrained with `num_steps=128`, do you expect this horizon to move? Argue from the product of state Jacobians in Equation 8.24.

2. Retrain with `num_steps` set to 128, measure the gradient against the lag on stretches of 256 validation tokens, and compare the result with your prediction.
3. Repeat the loss-by-position measurement of Section 8.5 for the retrained model. Does the loss now keep improving beyond the third token of context?

8.7 Decoding and Generation

The language models of this chapter have been judged by how well they predict the next token. To write text, a model must also choose one token at each step from the distribution it returns, and this choice matters more than one might expect: different rules make the same model write very different text. The simplest rule, tried in Section 8.5, fell short: always taking the most probable token tends to run into loops. How, then, should the choice be made?

The rule that turns next-token distributions into sequences is called *decoding* (Graves 2013; Holtzman et al. 2020), and decoding rules come in two families. *Maximization* searches for a sequence of high probability under the model. *Sampling* draws each token from the model's distribution, after cutting off the part of it that cannot be trusted. We develop both in turn. All of these rules see the model only through its next-token distribution, so they apply unchanged to any language model, recurrent or not.

8.7.1 The Decoding Problem

A trained language model supplies conditionals: writing $x_{<t}$ for the prefix $x_1, \dots, x_{t-1}$, it turns any prefix into a distribution $P(x_t \mid x_{<t})$ over the next token. By the chain rule of probability Equation 8.2, these conditionals assign every sequence the probability $P(x_1, \dots, x_T) = \prod_{t=1}^{T} P(x_t \mid x_{<t})$.

Decoding asks the inverse question: given the conditionals, which sequence should we output? The obvious answer is the *most probable* one. Unfortunately, finding it is out of the question. With vocabulary $\mathcal{V}$ and T tokens to generate there are $|\mathcal{V}|^T$ candidates. For our 1,027 tokens and a 50-token continuation this is $1027^{50} > 10^{150}$, far too many to enumerate, let alone compare. A practical decoder therefore extends only a few prefixes at each step and must decide which ones to keep.

Even if we could find it, the most probable sequence is often not what we want. When the input admits only a narrow set of acceptable outputs, as in transcribing speech, a high-probability output is a good target. When many continuations are acceptable, as in writing a story, each of them is individually improbable, and a model that has learned this spreads its probability across them. Insisting on the single most probable output then throws away the variety the model has learned. Sampling keeps it, because it draws the output from the model's distribution instead of maximizing it.

Figure 8.10 sorts the decoding rules into these two families. Maximization methods, greedy decoding and beam search, are common when the acceptable outputs are few. Sampling methods, which first reshape or truncate the distribution, are the default for open-ended generation such as dialogue and stories.

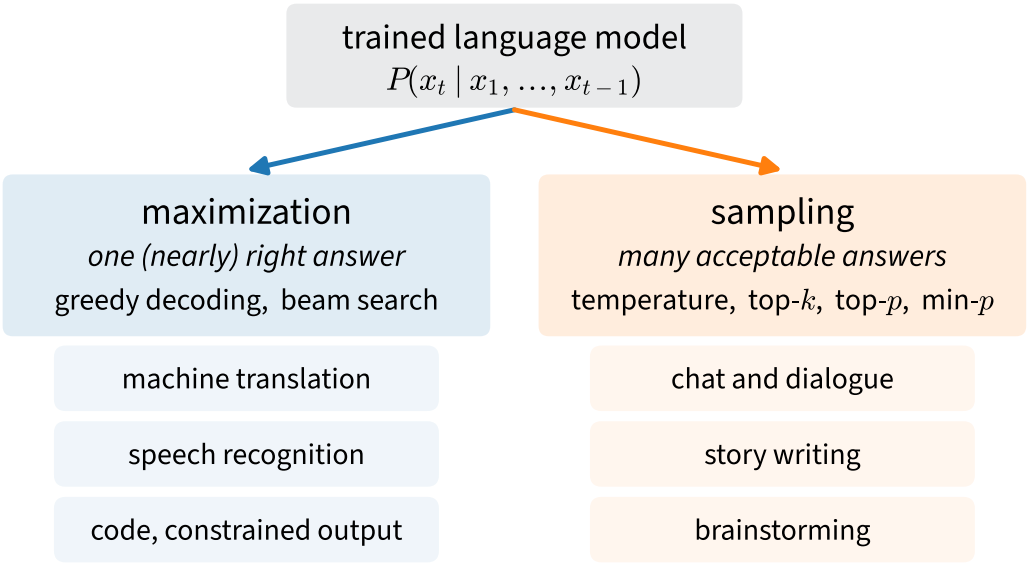


Figure 8.10: Two families of decoding rules. Searching for high-probability sequences suits tasks with few acceptable outputs; sampling preserves diversity in open-ended generation.

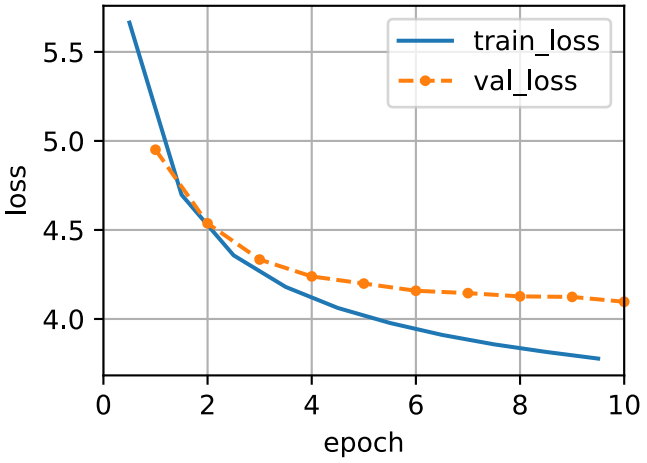
8.7.1.1 A Model to Decode From

We decode from the concise RNN language model of Section 8.5.5: an embedding of dimension 512, a recurrent layer with 128 hidden units, dropout on the embeddings and the hidden states, and an output layer over the 1,027 BPE tokens of *The Time Machine*, trained for ten epochs on windows of 32 tokens with gradient clipping. We retrain it with the same settings and print its validation perplexity, which should be close to the value reached there.

```
data = d2l.TimeMachine(batch_size=1024, num_steps=32)
```

```
rnn = d2l.RNN(num_inputs=512, num_hiddens=128)
model = d2l.RNNLM(rnn, vocab_size=len(data.vocab), lr=4, dropout=0.5,
                  emb_dropout=0.1)
```

```
trainer = d2l.Trainer(max_epochs=10, gradient_clip_val=1, num_gpus=1)
trainer.fit(model, data)
print(f'validation perplexity {math.exp(trainer.history["val_loss"][-1]):.1f}')
```




```
validation perplexity 60.1
```

Every decoding rule in this section sees the model through one interface: given the token ids so far, return the logits of the next token. We wrap the trained network in a function `step_fn` of this form. It returns a NumPy vector, so that the decoding rules can be written once, in NumPy, for all frameworks. The interface carries no hidden state, and this is deliberate: any model that predicts next tokens can fill it, including one that has no fixed-size state to carry.

```
model.eval()  # Dropout off

def step_fn(ids):  # Token ids in, NumPy logits for the next token out
    with torch.inference_mode():
        logits = model(d2l.tensor([ids], device=d2l.try_gpu()))
    return d2l.numpy(logits)[0, -1]
```

In return, each call runs the network over the *entire* prefix to produce one vector of logits. For a recurrent network this is wasteful, and we count the waste at the end of the section.

8.7.2 Greedy Decoding

The simplest decoder commits, at every step, to the most probable token:

$$x_t = \operatorname*{argmax}_{x \in \mathcal{V}} P(x \mid x_{<t}). \tag{8.30}$$

Score, take the argmax, append, repeat. Greedy decoding costs one model evaluation per token and is deterministic: the same prefix always yields the same continuation. To print continuations, `show` decodes token ids and quotes the text, with line breaks written as `\n` and unprintable characters replaced by U+FFFD.

```
def show(ids):  # Decoded text as a literal; control characters -> U+FFFD
    text = data.tokenizer.decode(ids)
    return repr(''.join(c if c.isprintable() or c == '\n' else '\ufffd'
```

```
                        for c in text))

    def decode_greedy(text, num_tokens=50):
        ids = data.tokenizer.encode(text)
        for _ in range(num_tokens):
            ids.append(int(step_fn(ids).argmax()))
        return ids

    for text in ('The Time Traveller', 'It seemed to me that'):
        d2l.print_wrapped(show(decode_greedy(text)))
```

```
"The Time Traveller.\n\n'I was\nthe Time Traveller.\n\n'I was\nthe Time
    Traveller.\n\n'I was\nthe Time Traveller.\n\n'I was\nthe Time
    Traveller.\n\n'I was\nthe Time Traveller"
"It seemed to me that\nthe\nthe Time Traveller.\n\n'I was\nthe Time
    Traveller.\n\n'I was\nthe Time Traveller.\n\n'I was\nthe Time
    Traveller.\n\n'I was\nthe Time Traveller.\n\n'"
```

Both continuations end in a loop, as the greedy sample of Section 8.5 did. Two flaws are built into the rule. The first is *myopia*: a sequence of locally best tokens need not be the best sequence. Suppose the model gives the first token the probabilities $P(A) = 0.5$ and $P(B) = 0.4$. After A the model is unsure, and its most probable second token has probability 0.3; after B it is confident, with 0.9 for its favorite. Greedy decoding picks A and ends with a sequence of probability $0.5 \times 0.3 = 0.15$, whereas starting with B gives $0.4 \times 0.9 = 0.36$. Greedy decoding cannot see that the more probable first token leads into an uncertain second step.

The second flaw is the loop itself. Greedy decoding feeds its own argmax back in as context, and nothing in it is random: the next token is a function of the hidden state, and the next state is a function of the state and that token. Once the state has settled into a cycle, no chance event can dislodge it. Is this a defect of our small model? Unfortunately not. Holtzman et al. (2020) documented this *degeneration* in language models about a thousand times larger than ours: maximization-based decoding of open-ended text produces output that is repetitive and measurably unlike human text, however good the model.

8.7.3 Beam Search

Greedy decoding commits immediately; exhaustive search never commits at all. *Beam search* lies between them, with a single setting, the *beam width* w: it keeps the w most probable prefixes alive at every step. At the first step it selects the w most probable tokens. After that, each of the w surviving prefixes is extended by every token in the vocabulary, the resulting $w \cdot |\mathcal{V}|$ candidates are scored by their cumulative log-probability, and the best w survive. Greedy decoding is the case $w = 1$. Exhaustive search is the limit in which w exceeds the number of candidates and nothing is ever pruned.

Figure 8.11 traces an example with $w = 2$ over the vocabulary $\mathcal{V} = \{A, B, C, D, E\}$ and three steps. Every probability is conditional on the given prefix, which we leave out of the notation. Suppose that the two most probable first tokens are A and C; they form the initial beam. At the second step we compute, for every $x' \in \mathcal{V}$, the scores of the ten extensions

$$P(A, x') = P(A)\, P(x' \mid A),$$
$$P(C, x') = P(C)\, P(x' \mid C),$$

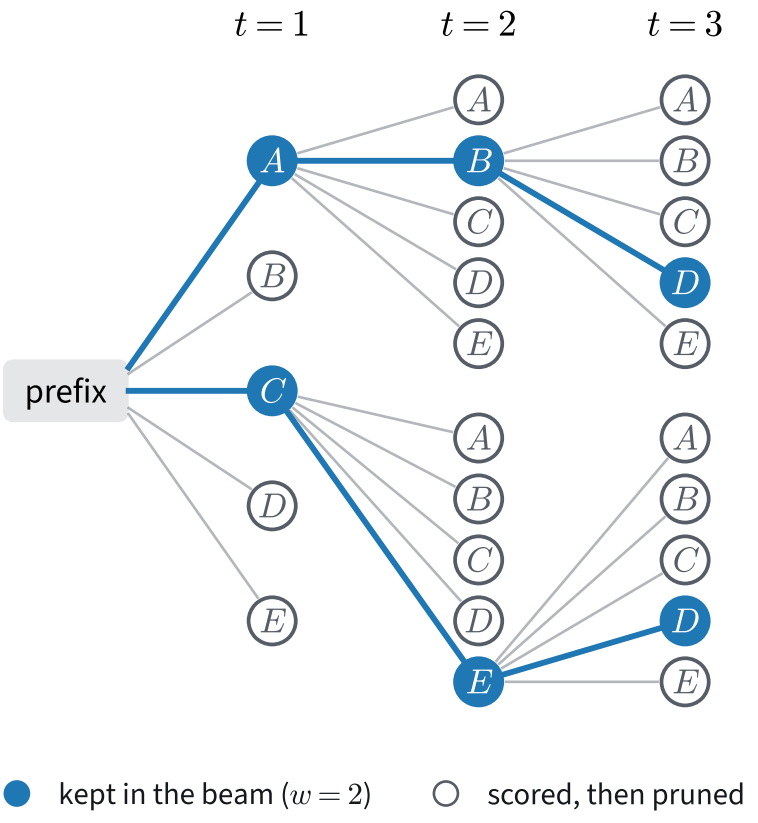


Figure 8.11: Beam search with beam width 2 over a five-token vocabulary. At every step each surviving prefix is expanded over the whole vocabulary and only the two highest-scoring prefixes are kept; the survivors here are A and C, then AB and CE, then ABD and CED.

and keep the two largest, say for AB and CE. The third step repeats the calculation from these survivors and keeps, say, ABD and CED. Instead of the $5^3 = 125$ sequences of exhaustive search, beam search scored $5 + 10 + 10 = 25$. In general it costs $O(w|\mathcal{V}|T)$ for T steps, w times as much as greedy decoding and exponentially less than exhaustive search.

One correction is needed before finished candidates of different lengths can be compared. The log-probability of a token is never positive, so the score of a sequence can only fall as it grows, and comparing raw scores favors short candidates. This matters once an end-of-sequence token lets some candidates stop early. The standard remedy is *length normalization* (Wu et al. 2016): rank a candidate that adds T tokens to a prefix of t_0 tokens by

$$\frac{1}{T^\beta} \sum_{t=t_0+1}^{t_0+T} \log P(x_t \mid x_{<t}), \tag{8.31}$$

where the exponent β, typically around 0.75, interpolates between no correction ($\beta = 0$) and the average log-probability per token ($\beta = 1$). In the code below it is the argument `alpha`, the name used in Wu et al. (2016) .

The implementation below follows this procedure. Finished candidates, those that produced `eos_id`, pass from round to round unchanged; live ones are expanded and rescored. Since the best w candidates contain at most w continuations of any single prefix, it suffices to expand each prefix by its w most probable tokens rather than by all $|\mathcal{V}|$ of them.

```
def beam_search(step_fn, prefix, num_tokens, beam_size=4,
                alpha=0.75, eos_id=None):
    """Length-normalized beam search (score = log P / len^alpha).
    Returns candidates sorted best-first; [0][1] is the argmax sequence."""
    def log_softmax(logits):
        logits = [float(l) for l in logits]
        m = max(logits)
        lse = m + math.log(sum(math.exp(l - m) for l in logits))
        return [l - lse for l in logits]
    score = lambda logp, ids: logp / max(1, len(ids) - len(prefix))**alpha
    beams = [(0.0, list(prefix), False)]  # (log-prob, sequence, finished)
    for _ in range(num_tokens):
        if all(done for _, _, done in beams):
            break
        candidates = [b for b in beams if b[2]]  # Finished pass through
        for logp, ids, _ in (b for b in beams if not b[2]):
            logprobs = log_softmax(step_fn(ids))
            best = sorted(range(len(logprobs)),
                          key=lambda i: -logprobs[i])[:beam_size]
            candidates += [(logp + logprobs[i], ids + [i], i == eos_id)
                           for i in best]
        beams = sorted(candidates, key=lambda b: score(b[0], b[1]),
                       reverse=True)[:beam_size]
    return sorted([(score(logp, ids), ids) for logp, ids, _ in beams],
                  reverse=True)
```

We decode the two prefixes from before with $w = 1$, which reproduces greedy decoding, and with $w = 4$, and print the score Equation 8.31 of the winning candidate. Our text has no

end-of-sequence token, so every candidate has 30 tokens and the length correction changes no ranking here: the score is the log-probability divided by $30^{0.75} \approx 12.8$.

```
for text in ('The Time Traveller', 'It seemed to me that'):
    prefix = data.tokenizer.encode(text)
    for w in (1, 4):
        score, ids = beam_search(step_fn, prefix, 30, beam_size=w)[0]
        d2l.print_wrapped(f'w={w}, score {score:.2f}: {show(ids)}')
```

```
w=1, score -3.50: "The Time Traveller.\n\n'I was\nthe Time Traveller.\n\n'I
    was\nthe Time Traveller.\n\n'I was\nthe Time Traveller"
w=4, score -3.17: "The Time Traveller.\n\n'Then I\nsaid.\n\n'Then
    I\nsaid.\n\n'Then I\nsaid"
w=1, score -3.55: "It seemed to me that\nthe\nthe Time Traveller.\n\n'I
    was\nthe Time Traveller.\n\n'I was\nthe Time Traveller.\n\n'"
w=4, score -3.58: "It seemed to me that the\nsharacter.\n\n'Then
    I\nsaid.\n\n'Then I\nsaid.\n\n"
```

The wider beam usually finds continuations that the model scores higher than the greedy ones, but it cannot guarantee it: beam search is still an approximate search, and it can prune the prefix that greedy decoding would have completed. Whether the higher-scoring text is also *better* text is a separate question. In large language models, beam search on open-ended text produces output far more repetitive than human text (Holtzman et al. 2020), and we measure below how our maximizers compare with sampling. Where the acceptable outputs are few, as in speech recognition and machine translation, beam search with a small width, typically between 4 and 10, is the standard decoder (Sutskever et al. 2014; Wu et al. 2016).

8.7.4 Sampling

If we trust the model's distribution, the natural way to generate is to sample from it: draw $x_t \sim P(x_t \mid x_{<t})$, append, repeat. Text produced this way is distributed as the model believes text should be, with the variety that maximization throws away. The problem lies in the tail. Our model spreads small probabilities over a thousand tokens, a large model over a hundred thousand. Each unlikely token is negligible on its own, but their combined probability at each step is not. Sampling therefore regularly picks a token that the model itself considers nearly impossible, and one absurd token, fed back as context, can derail everything after it. This is the compounding of errors in a rollout (Section 8.1), with tokens in place of numbers. Holtzman et al. (2020) call the culprit the *unreliable tail*: the model's estimates are trustworthy for its most probable tokens and mostly noise far down the ranking. The fix is to reshape or truncate the distribution before sampling, and each of the settings below does this in its own way.

Temperature divides every logit o_x by $\tau > 0$ before the softmax:

$$P_\tau(x \mid x_{<t}) = \frac{\exp(o_x/\tau)}{\sum_{x' \in \mathcal{V}} \exp(o_{x'}/\tau)} \propto P(x \mid x_{<t})^{1/\tau}. \tag{8.32}$$

Values of τ below 1 sharpen the distribution, down to greedy decoding as $\tau \to 0$, and values above 1 flatten it toward uniform. Temperature is a blunt tool. It reshapes the head and the tail together, so cooling the distribution enough to suppress the tail also removes legitimate variety at the head.

Top-k truncates by count: keep the k most probable tokens, renormalize, and sample (Fan et al. 2018; Radford et al. 2019). **Top-p** (or *nucleus*) sampling truncates by probability mass: keep the smallest set of most probable tokens whose total probability reaches p, for example $p = 0.9$ (Holtzman et al. 2020). The difference matters because the model's confidence changes from step to step. Where the model is nearly certain of the next token, a fixed $k = 20$ admits nineteen poor options; where dozens of words are plausible, the same k may cut off real variety. Top-p adapts: it keeps few tokens when the head of the distribution is heavy and many when it is flat. It has its own failure on peaked distributions: if the top token holds 0.95 of the mass, top-p with $p = 0.9$ keeps one token and acts greedily even where the runner-up was a good choice.

Min-p sets the cutoff relative to the model's confidence (Nguyen et al. 2025): keep every token whose probability is at least p_{min} times that of the most probable token, for example $p_{\text{min}} = 0.05$. When the model is sure, the bar is high and few tokens pass; when it is unsure, the bar drops and more of them survive. Nguyen et al. (2025) find that this relative rule degrades less than top-p at high temperatures, where an absolute rule admits much of the flattened tail.

8.7.4.1 A Unified Sampler

All four settings act on the same object, the next-token distribution, so one function can implement them. `sample_next` turns a vector of logits into a token id. At temperature 0 it returns the argmax, which is greedy decoding. Otherwise it computes P_τ of Equation 8.32, sorts the tokens by probability, and applies the optional top-k, top-p, and min-p cutoffs. Each cutoff only decides how many tokens of the sorted order to keep, and the function keeps the smallest of these numbers. It then renormalizes the kept probabilities and draws one token. It takes a NumPy random generator `rng` and creates a fresh one if none is given.

```
def sample_next(logits, temperature=1.0, k=None, p=None, min_p=None,
                rng=None):
    """Draw a token id from 1-D NumPy logits; temperature 0 is greedy.
    Top-k, top-p, and min-p truncate the temperature-scaled distribution."""
    logits = np.asarray(logits, dtype=np.float64)
    if temperature == 0:
        return int(logits.argmax())
    probs = np.exp((logits - logits.max()) / temperature)
    order = np.argsort(-probs)              # token ids, most probable first
    probs = probs[order] / probs.sum()
    keep = len(probs)
    if k is not None:                       # top-k: a fixed count
        keep = min(keep, k)
    if p is not None:                       # top-p: smallest head of mass p
        keep = min(keep, int(np.searchsorted(probs.cumsum(), p)) + 1)
    if min_p is not None:                   # min-p: a fraction of the top
        keep = min(keep, int((probs >= min_p * probs[0]).sum()))
    rng = np.random.default_rng() if rng is None else rng
    return int(order[rng.choice(keep, p=probs[:keep] / probs[:keep].sum())])
```

`generate` is the loop we have been writing by hand: query `step_fn`, choose with `sample_next` (extra keyword arguments pass through to it), append, and stop early if the end-of-sequence token `eos_id` appears. Open-ended generation has no such token, so our examples run for a fixed number of tokens.

```
def generate(step_fn, prefix, num_tokens, eos_id=None, **strategy):
    """Extend prefix by up to num_tokens ids, each chosen by sample_next
    from step_fn(ids), the next-token logits; stop early at eos_id."""
    ids = list(prefix)
    for _ in range(num_tokens):
        ids.append(sample_next(step_fn(ids), **strategy))
        if eos_id is not None and ids[-1] == eos_id:
            break
    return ids
```

A toy distribution makes the cutoffs concrete. Five tokens have probabilities 0.45, 0.25, 0.15, 0.10, and 0.05. Top-k with $k = 2$ can only emit tokens 0 and 1. Top-p with $p = 0.8$ keeps three tokens, since the cumulative probability first reaches 0.8 at the third (0.85). Min-p with $p_{\text{min}} = 0.5$ keeps the tokens with probability at least $0.5 \times 0.45 = 0.225$, again tokens 0 and 1. We draw 1,000 tokens under each rule and list the tokens that occur.

```
logits = np.log([0.45, 0.25, 0.15, 0.10, 0.05])
print('greedy:', sample_next(logits, temperature=0))
for cutoff in (dict(k=2), dict(p=0.8), dict(min_p=0.5)):
    drawn = {sample_next(logits, **cutoff) for _ in range(1000)}
    print(cutoff, 'tokens drawn:', sorted(drawn))
```

```
greedy: 0
{'k': 2} tokens drawn: [0, 1]
{'p': 0.8} tokens drawn: [0, 1, 2]
{'min_p': 0.5} tokens drawn: [0, 1]
```

Each rule draws exactly the tokens it keeps: tokens 0 and 1 under top-k and min-p, tokens 0 to 2 under top-p.

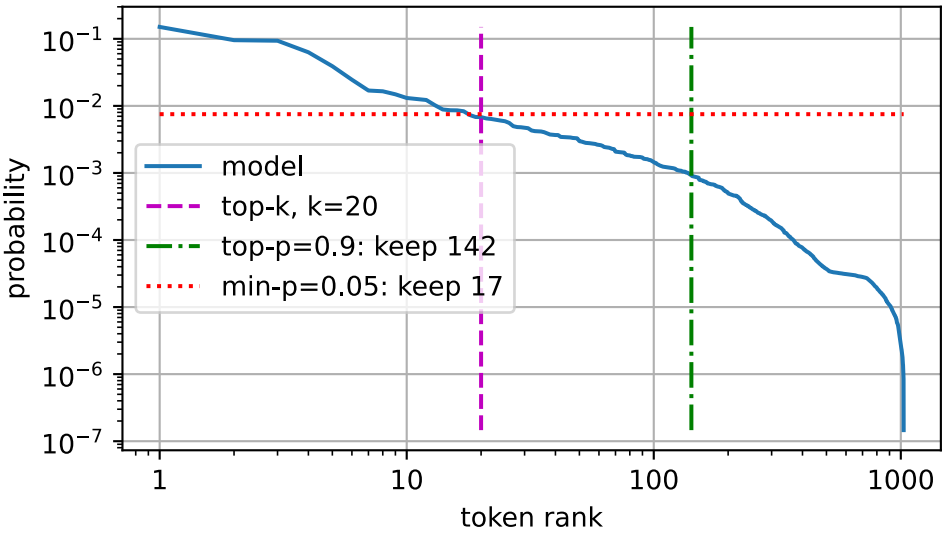
8.7.4.2 Cutoffs on a Trained Model

What do the cutoffs do to a real distribution? We compute our model's next-token distribution after "The Time Traveller", sort it by decreasing probability, and plot it on logarithmic axes. On this plot each rule is a single line. Top-k is a vertical line at rank k, wherever the probabilities lie. Top-p is a vertical line at the rank where the cumulative probability reaches p, so its position depends on the shape of the head. Min-p is a *horizontal* line at p_{min} times the top probability: every token above it is kept, and the line moves up and down with the model's confidence. Everything to the right of the vertical lines, or below the horizontal one, is the tail that truncation removes.

```
def next_probs(text):  # Next-token probabilities after text, sorted
    logits = step_fn(data.tokenizer.encode(text))
    probs = np.exp(logits - logits.max())
    return np.sort(probs / probs.sum())[::-1]

def cutoffs(probs, p=0.9, min_p=0.05):  # Tokens kept by top-p and min-p
    return (int(np.searchsorted(probs.cumsum(), p)) + 1,
            int((probs >= min_p * probs[0]).sum()))
```

```
probs = next_probs('The Time Traveller')
n_p, n_m = cutoffs(probs)
lo, hi = probs[-1], probs[0]
d2l.plot([np.arange(1, len(probs) + 1), [20, 20], [n_p, n_p],
          [1, len(probs)]], [probs, [lo, hi], [lo, hi], [0.05 * hi] * 2],
         xlabel='token rank', ylabel='probability', xscale='log',
         yscale='log', figsize=(5.5, 3),
         legend=['model', 'top-k, k=20', f'top-p=0.9: keep {n_p}',
                 f'min-p=0.05: keep {n_m}'])
```



On the logarithmic axes the curve declines steadily, from a top probability of about 0.2 to about 10^{-3} near rank 100, and falls faster still over the last few hundred ranks. Most of the vocabulary lies in this tail of tokens with tiny probabilities, the tail that all three rules cut. To compare the rules, we also apply them after the shorter prefix “The Time”, which the novel continues with ” Traveller” in 11 of its 13 occurrences, so the model should be nearly sure of the next token. We count the tokens each rule keeps after both prefixes.

```
for text in ('The Time Traveller', 'The Time'):
    probs = next_probs(text)
    n_p, n_m = cutoffs(probs)
    print(f'{text!r}: top probability {probs[0]:.2f}; kept: top-k 20, '
          f'top-p {n_p}, min-p {n_m}')
```

```
'The Time Traveller': top probability 0.15; kept: top-k 20, top-p 142, min-p 17
'The Time': top probability 0.78; kept: top-k 20, top-p 2, min-p 2
```

Top-k keeps 20 tokens whatever the distribution. Both other rules tighten when the model is sure: after “The Time” the top probability is high, both keep far fewer tokens, and min-p keeps only a handful. They differ when the model is unsure. After “The Time Traveller”, top-p must reach far into the tail to collect 90% of the mass and keeps well over a hundred tokens, most of them with probability below one percent. Min-p looks only at the top probability: it keeps the tokens within a factor of twenty of the best, a few dozen at most. Which of the

two is right, the counts cannot say.

8.7.4.3 Strategies Compared on One Prefix

We now read the text that each setting produces, starting with temperature alone at $\tau = 0.5$, 1, and 2.

```
prefix = data.tokenizer.encode('The Time Traveller')
for tau in (0.5, 1.0, 2.0):
    out = generate(step_fn, prefix, 50, temperature=tau)
    d2l.print_wrapped(f'tau={tau}: {show(out)}')
```

```
tau=0.5: "The Time Traveller.\n\n'I was a little people\nthe Befification
    of\ntheir\nbeen to\nmyself, and\nthe\nthese of the\ntheir hazy and
    disapposition"
tau=1.0: 'The Time Traveller, my buildly clamp facesmeials.\n\n\nBetmitting
    of point his broken rights; her Lurned to me and course people; full
    fast with my f'
tau=2.0: 'The Time Traveller you any about it chitt wat towards Weena my
    back;es spillish im into◆ardeping cur saw evenable San tried: that◆ chan
    round sl diffense lif heattallV◆ her of Nood far on whair'
```

At $\tau = 0.5$ the text stays close to the model's most probable tokens. At $\tau = 1$ it is more varied and occasionally strays into invented words. At $\tau = 2$ the flattened distribution draws so many tail tokens that the text stops being English; the tokens may even combine into bytes that do not form valid characters. Next we run the main strategies on the same prefix, each with its typical setting.

```
strategies = {'greedy': dict(temperature=0), 'tau=1': dict(),
              'top-k=20': dict(k=20), 'top-p=0.9': dict(p=0.9),
              'min-p=0.05': dict(min_p=0.05)}
for name, strategy in strategies.items():
    out = generate(step_fn, prefix, 50, **strategy)
    d2l.print_wrapped(f'{name:>10}: {show(out)}')
```

```
    greedy: "The Time Traveller.\n\n'I was\nthe Time Traveller.\n\n'I
    was\nthe Time Traveller.\n\n'I was\nthe Time Traveller.\n\n'I was\nthe
    Time Traveller.\n\n'I was\nthe Time Traveller"
     tau=1: 'The Time Traveller. Asult\nall future was looked\nIought my
    mind.\nthe Pals seemed and the hyance had late I heard fear
    how\nthingwas\nMachine trinkminated. The inst'
  top-k=20: "The Time Traveller, as I rescention. And with a queak in an
    excepted for this crows of men.\n\n'Now.\n\n'Then\nI care into\nwhich
    my"
 top-p=0.9: 'The Time Traveller made my fram of the bir intent Man had
    travest of the new\nside an confus of a tone was\nfaceleamined count;
    but\nthe whole poid he:\nin'
min-p=0.05: 'The Time Traveller. Then the Psychologist\nbegard, with
    the\nwell the\natight, I took I felt that the Editor, as I\nhad
    takes\ncame. Had it to\na lay at these in their'
```

Our small model cannot make any of these continuations good, and one sample per rule is thin evidence. Greedy decoding is deterministic: rerunning the cell reproduces its text

exactly, while every sampler gives new text each time. Only greedy decoding loops. Among the samplers these lines show no ranking, since invented words appear with and without truncation. The truncated samplers are designed to keep the variety of sampling and cut the tail on which pure sampling trips, and a temperature combined with top-p or min-p is a common default for open-ended generation.

8.7.5 Evaluation and Cost

8.7.5.1 The Quality of Generated Text

How can we decide, beyond reading samples, which strategy writes better text? Perplexity is of little help. It evaluates the *model*: how well its conditionals predict text that people wrote. It says nothing about a particular generated sample, and scoring a sample by its probability under the model would reward maximization by definition. A strategy is not better because its output is more probable; sampling deviates from the most probable output on purpose.

Automated statistics can still catch specific failures. A simple one for repetition is *distinct-3*, the fraction of distinct 3-grams among all 3-grams of a continuation: it is 1 when no three consecutive tokens ever recur and close to 0 for text stuck in a loop. We decode 100 tokens after 20 random six-token prefixes from the validation text with greedy decoding, beam search with $w = 4$, and the samplers of the previous section, and report two averages: the log-probability per generated token under the model, and distinct-3.

```
def log_prob(ids, start):  # Mean log-probability of ids[start:]
    lps = []
    for t in range(start, len(ids)):
        logits = step_fn(ids[:t])
        lse = logits.max() + np.log(np.exp(logits - logits.max()).sum())
        lps.append(logits[ids[t]] - lse)
    return np.mean(lps)

def distinct(ids, n=3):  # Fraction of distinct n-grams in ids
    grams = [tuple(ids[i:i + n]) for i in range(len(ids) - n + 1)]
    return len(set(grams)) / len(grams)

val = d2l.numpy(data.X[data.num_train:data.num_train + data.num_val])
rows = np.random.default_rng().choice(len(val), 20, replace=False)
prefixes = [[int(i) for i in val[r, :6]] for r in rows]
decoders = {
    'greedy': lambda pre: generate(step_fn, pre, 100, temperature=0),
    'beam, w=4': lambda pre: beam_search(step_fn, pre, 100, beam_size=4)[0][1],
    'tau=1': lambda pre: generate(step_fn, pre, 100),
    'top-k=20': lambda pre: generate(step_fn, pre, 100, k=20),
    'top-p=0.9': lambda pre: generate(step_fn, pre, 100, p=0.9),
    'min-p=0.05': lambda pre: generate(step_fn, pre, 100, min_p=0.05)}
for name, decode in decoders.items():
    outs = [decode(pre) for pre in prefixes]
    print(f'{name:>10}: log-prob per token '
          f'{np.mean([log_prob(o, 6) for o in outs]):.2f}, distinct-3 '
          f'{np.mean([distinct(o[6:]) for o in outs]):.2f}')
```

```
   greedy: log-prob per token -1.60, distinct-3 0.12
beam, w=4: log-prob per token -1.39, distinct-3 0.13
```

```
     tau=1: log-prob per token -4.04, distinct-3 1.00
  top-k=20: log-prob per token -2.87, distinct-3 0.99
 top-p=0.9: log-prob per token -3.57, distinct-3 0.99
min-p=0.05: log-prob per token -2.89, distinct-3 0.99
```

The two averages disagree. The two maximizers produce continuations with a clearly higher log-probability per token than any of the samplers. Beam search is usually ahead of greedy decoding, by a margin that varies from run to run and occasionally reverses. Yet the samples are far more varied: their distinct-3 is close to 1, while the maximizers repeat most of their 3-grams. Maximization raises the probability of the text and lowers its variety, the degeneration described by Holtzman et al. (2020) on a small scale.

Among the samplers, variety does not separate the rules, but probability does. Pure sampling at $\tau = 1$ has the lowest log-probability per token, about -4. The three truncated samplers lie between it and the maximizers, and their continuations are as varied as those of pure sampling. Top-p at 0.9 stays nearest to pure sampling, which is what the counts above suggested, since it keeps the most tokens; top-k and min-p cut deeper, and their text is about a nat per token more probable. Cutting the tail does what it was designed to do.

Such statistics detect particular failures; they do not say whether the text is good. The final judge of open-ended generation is a human reader: can people tell the model's text from text that people wrote, and which do they prefer? Tasks with a reference answer, such as translation, are easier to score automatically, by the overlap between the output and reference translations. In short, perplexity selects the model, and human judgment, or a proxy for it, selects the decoding strategy.

8.7.5.2 The Cost of Generation

We built `step_fn` without a state so that it would fit any model, and so it runs the network over the whole prefix for every token it generates. A continuation of T tokens thus costs $O(T^2)$ updates of the recurrent cell. Generation is also *sequential*: during training, all positions of a known sequence are processed together, but token $t + 1$ cannot be scored before token t has been chosen. The sequential part cannot be avoided. The quadratic part, for a recurrent network, is pure waste. The hidden state already summarizes the prefix, so carrying it forward, as `predict` in Section 8.5 does, makes each new token cost one cell update, independent of the length of the prefix.

This constant cost and memory per generated token is what a fixed-size state provides, and a reason for the interest in recurrent language models. The same state is the model's limit: whatever it remembers of the prefix must fit into 128 numbers (Section 8.4.2.1). No decoding rule can put into the text what the model's distribution does not contain. The decoders, however, never looked inside the model, and they will serve unchanged when a better one supplies the distribution.

With this the first part of the book ends, and it ends on a limit. Our last model predicts text one token at a time at a constant cost, and it makes use of about three tokens of what it has read (Section 8.5), because its gradients reach hardly further (Section 8.6). There are two ways out. One keeps the fixed-size state and redesigns the recurrence so that the gradient survives (Chapter 12). The other gives up the fixed size: it keeps every past token and lets the model look back at all of them, at a cost that grows with the length of the text. This is attention (Chapter 10), and today's language models are built on it. Whichever is chosen, the tokenizer, the bits per byte, and the decoders of this chapter carry over as they are.

8.7.6 Summary

Decoding turns a language model's next-token distributions into sequences. Maximization targets the most probable output: greedy decoding takes the most probable token at each step and is myopic, and beam search keeps the w best prefixes at w times the cost. Maximization suits tasks with few acceptable outputs, but on open-ended text its output is repetitive.

Sampling draws from the model's distribution, reshaped by temperature Equation 8.32 and truncated by top-k (a fixed count), top-p (a fixed probability mass), or min-p (a fixed fraction of the top probability). On our model the truncated samplers write text as varied as pure sampling and more probable. Perplexity evaluates the model, not its samples, and generating naively re-runs the model over the whole prefix for every new token, which a carried recurrent state avoids.

8.7.7 Exercises

1. code **Repetition penalty.** The *repetition penalty* of Keskar et al. (2019) discourages tokens that already appear in the sequence by dividing each such token's logit by a constant $\gamma > 1$. Add it to `sample_next` so that it acts on the logits before both the greedy argmax and the temperature scaling. `generate` then has to pass the ids so far to `sample_next`. Decode `'The Time Traveller'` and `'It seemed to me that'` greedily with $\gamma = 1.2$.
    1. Does the penalty remove the repeating cycles? Report the distinct-3 of each continuation, computed with `distinct`, with and without the penalty.
    2. Adding the same constant to every logit leaves the softmax unchanged. Does it leave the penalized distribution unchanged? Work out what the penalty does to a token whose logit is negative.
    3. Which legitimate repetitions in English text does the penalty discourage, and what happens as γ grows large?
2. **Beam search properties.**
    1. Take beam width 2, the five-token vocabulary of Figure 8.11, and three steps. Assign conditional probabilities for which beam search does not return the most probable three-token sequence, and compute the probabilities of both sequences. What does your construction imply about how beam search can miss the most probable sequence?
    2. What is the smallest beam width that guarantees that beam search over T steps returns the most probable of all $|\mathcal{V}|^T$ sequences, whatever the model?
    3. Let $w \leq |\mathcal{V}|$, and assume that no end-of-sequence token occurs. Count the calls to `step_fn` that `beam_search` makes over T steps. Count also the calls that exhaustive search makes when it evaluates every prefix once. Evaluate both counts for $|\mathcal{V}| = 1{,}027$, $T = 50$, and $w = 4$.
3. code **Length normalization.** Run `beam_search` with $w = 4$ for up to 40 tokens after `'The Time Traveller'` and `'It seemed to me that'`. Pass the id of the newline character, `data.tokenizer.encode('\n')[0]`, as `eos_id`, and vary β (the argument `alpha`) over $\{0, 0.75, 1.5\}$. Compare the lengths and the total log-probabilities of the winning candidates, and explain the trend using Equation 8.31.
4. **Truncation under temperature.** `sample_next` applies every cutoff to the temperature-scaled distribution P_τ of Equation 8.32. Write the min-p condition $P_\tau(x \mid x_{<t}) \geq$

$p_{\text{min}} \max_{x'} P_\tau(x' \mid x_{<t})$ as a condition on the untempered probabilities. For the five-token distribution 0.45, 0.25, 0.15, 0.10, 0.05, compute the sets that top-k with $k = 2$, top-p with $p = 0.85$, and min-p with $p_{\text{min}} = 0.5$ keep at $\tau = 2$. Which of the three rules keeps the same set at every temperature?

5. code **Constrained sampling.** Ban every token whose text contains the letter "e" by setting its logit to $-\infty$ in a wrapper around `step_fn`. Generate 50 tokens greedily after `'The Time Traveller'` with and without the ban. Report the probability mass that the ban removes, averaged over the steps of the banned continuation, and describe how the two continuations differ.

    1. Why is banning a *word* harder than banning a token under a subword tokenizer, and what can go wrong at the boundary between two tokens?
    2. Grammar and schema constraints are usually enforced by this same per-step masking, with a parser deciding at each step which tokens keep the output valid. Some constraints span several tokens, such as forcing a given phrase to appear somewhere in the output. What does keeping several prefixes alive, as beam search does, add over masking one token at a time for such a constraint?

6. code **Top-p versus min-p.** ▣ At $\tau = 1.5$, sweep p for top-p and p_{min} for min-p over a grid of values. For each setting, generate 50-token continuations of five prefixes of your choice and record two averages over the generated tokens: the number of tokens the rule keeps, and the log-probability of the drawn token under the untempered model. Plot the second average against the first, one curve per rule. Which rule draws more probable tokens at the same average number of kept tokens? Compare your finding with those of Nguyen et al. (2025) .

7. code **Failure taxonomy.** ▣ Choose 20 prefixes and decode a 50-token greedy continuation of each.

    1. Sort the failures into a taxonomy of your own, with categories such as a repeating cycle or a drift away from the prefix's topic, and report the frequency of each category with one example.
    2. Decode one continuation of each prefix with min-p sampling at $p_{\text{min}} = 0.05$ and classify its failures with the same taxonomy, adding categories where needed. Which categories does sampling remove, and which does it introduce?
    3. Which of the automated statistics of Section 8.7.5.1 would detect each category, and which categories need a human judge?

    Adapted from Stanford CS224n, Assignment 4, Part 2(b).

8. code **Constant cost per token.** ▣ `step_fn` re-runs the network over the entire prefix for every generated token, so a T-token continuation costs $O(T^2)$ cell updates (Section 8.7.5.2).

    1. Write a stateful replacement for `step_fn` that keeps the hidden state of `model.rnn` and consumes only the newest token, so that each call performs one cell update. Verify that `generate` produces the same greedy continuation with both versions.
    2. Time `generate` for 50, 200, and 800 tokens with both versions. Estimate from the ratios how each time grows with the number of tokens, and explain any departure from the quadratic and linear growth that the count of cell updates predicts.
    3. Why does the same replacement not carry over directly to `beam_search`, and what bookkeeping would make it work?

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.

Alsallakh, Bilal, Narine Kokhlikyan, Vivek Miglani, Jun Yuan, and Orion Reblitz-Richardson. 2020. "Mind the PAD – CNNs Can Develop Blind Spots." *ArXiv:2010.02178*. https://arxiv.org/abs/2010.02178.

Arora, Sanjeev, Zhiyuan Li, and Kaifeng Lyu. 2019. "Theoretical Analysis of Auto Rate-Tuning by Batch Normalization." *International Conference on Learning Representations*. https://arxiv.org/abs/1812.03981.

Arpit, Devansh, Stanisław Jastrzębski, Nicolas Ballas, et al. 2017. "A Closer Look at Memorization in Deep Networks." *International Conference on Machine Learning*, 233–42.

Ba, Jimmy Lei, Jamie Ryan Kiros, and Geoffrey E Hinton. 2016. "Layer Normalization." *ArXiv:1607.06450*. https://arxiv.org/abs/1607.06450.

Bay, Herbert, Tinne Tuytelaars, and Luc Van Gool. 2006. "SURF: Speeded up Robust Features." *European Conference on Computer Vision*, 404–17. https://doi.org/10.1007/11744023_32.

Bayes, Thomas. 1763. "An Essay Towards Solving a Problem in the Doctrine of Chances." *Philosophical Transactions of the Royal Society of London* 53: 370–418.

Belkin, Mikhail, Daniel Hsu, Siyuan Ma, and Soumik Mandal. 2019. "Reconciling Modern Machine-Learning Practice and the Classical Bias–Variance Trade-Off." *Proceedings of the National Academy of Sciences* 116 (32): 15849–54.

Bengio, Samy, Oriol Vinyals, Navdeep Jaitly, and Noam Shazeer. 2015. "Scheduled Sampling for Sequence Prediction with Recurrent Neural Networks." *Advances in Neural Information Processing Systems* 28. https://arxiv.org/abs/1506.03099.

Bengio, Yoshua, Réjean Ducharme, Pascal Vincent, and Christian Jauvin. 2003. "A Neural Probabilistic Language Model." *Journal of Machine Learning Research* 3 (Feb): 1137–55. https://jmlr.org/papers/v3/bengio03a.html.

Bengio, Yoshua, Nicholas Léonard, and Aaron Courville. 2013. "Estimating or Propagating Gradients Through Stochastic Neurons for Conditional Computation." *arXiv Preprint arXiv:1308.3432*.

Bengio, Yoshua, Patrice Simard, and Paolo Frasconi. 1994. "Learning Long-Term Dependencies with Gradient Descent Is Difficult." *IEEE Transactions on Neural Networks* 5 (2): 157–66. https://doi.org/10.1109/72.279181.

Beyer, Lucas, Olivier J. Hénaff, Alexander Kolesnikov, Xiaohua Zhai, and Aäron van den Oord. 2020. "Are We Done with ImageNet?" *ArXiv:2006.07159*. https://arxiv.org/abs/2006.07159.

Bishop, Chris M. 1995. "Training with Noise Is Equivalent to Tikhonov Regularization." *Neural Computation* 7 (1): 108–16. https://doi.org/10.1162/neco.1995.7.1.108.

Black, Fischer, and Myron Scholes. 1973. "The Pricing of Options and Corporate Liabilities." *Journal of Political Economy* 81: 637–54. https://doi.org/10.1086/260062.

Blum, Avrim, and Moritz Hardt. 2015. "The Ladder: A Reliable Leaderboard for Machine Learning Competitions." *Proceedings of the 32nd International Conference on Machine Learning*, Proceedings of machine learning research, vol. 37: 1006–14.

Bottou, Léon. 2010. "Large-Scale Machine Learning with Stochastic Gradient Descent." In *Proceedings of COMPSTAT'2010*. Springer. https://doi.org/10.1201/b11429-4.

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.

Brock, Andrew, Soham De, Samuel L. Smith, and Karen Simonyan. 2021. "High-Performance Large-Scale Image Recognition Without Normalization." *International Conference on Machine Learning*. https://arxiv.org/abs/2102.06171.

Chen, Stanley F., and Joshua Goodman. 1999. "An Empirical Study of Smoothing Techniques for Language Modeling." *Computer Speech & Language* 13 (4): 359–93. https://doi.org/10.1006/csla.1999.0128.

Chen, Tianqi, Mu Li, Yutian Li, et al. 2015. "MXNET: A Flexible and Efficient Machine Learning Library for Heterogeneous Distributed Systems." *ArXiv:1512.01274*. https://arxiv.org/abs/1512.01274.

Chen, Tianqi, Bing Xu, Chiyuan Zhang, and Carlos Guestrin. 2016. "Training Deep Nets with Sublinear Memory Cost." *arXiv Preprint arXiv:1604.06174*.

Chollet, François. 2017. "Xception: Deep Learning with Depthwise Separable Convolutions." *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. https://arxiv.org/abs/1610.02357.

Chu, Xiangxiang, Liang Li, and Bo Zhang. 2024. "Make RepVGG Greater Again: A Quantization-Aware Approach." *AAAI Conference on Artificial Intelligence*. https://arxi

v.org/abs/2212.01593.

Cover, Thomas M., and Peter E. Hart. 1967. "Nearest Neighbor Pattern Classification." *IEEE Transactions on Information Theory* 13 (1): 21–27.

Cubuk, Ekin D., Barret Zoph, Jonathon Shlens, and Quoc V. Le. 2020. "RandAugment: Practical Automated Data Augmentation with a Reduced Search Space." *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition Workshops*. https://arxiv.org/abs/1909.13719.

Cybenko, George. 1989. "Approximation by Superpositions of a Sigmoidal Function." *Mathematics of Control, Signals and Systems* 2 (4): 303–14. https://doi.org/10.1007/BF02551274.

Dalal, Navneet, and Bill Triggs. 2005. "Histograms of Oriented Gradients for Human Detection." *2005 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR'05)* 1: 886–93. https://doi.org/10.1109/cvpr.2005.177.

De Cock, Dean. 2011. "Ames, Iowa: Alternative to the Boston Housing Data as an End of Semester Regression Project." *Journal of Statistics Education* 19 (3). https://doi.org/10.1080/10691898.2011.11889627.

Deng, Jia, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. 2009. "Imagenet: A Large-Scale Hierarchical Image Database." *2009 IEEE Conference on Computer Vision and Pattern Recognition*, 248–55. https://doi.org/10.1109/cvpr.2009.5206848.

Der Kiureghian, Armen, and Ove Ditlevsen. 2009. "Aleatory or Epistemic? Does It Matter?" *Structural Safety* 31 (2): 105–12. https://doi.org/10.1016/j.strusafe.2008.06.020.

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.

Ding, Xiaohan, Xiangyu Zhang, Ningning Ma, Jungong Han, Guiguang Ding, and Jian Sun. 2021. "RepVGG: Making VGG-Style ConvNets Great Again." *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*. https://arxiv.org/abs/2101.03697.

Ding, Xiaohan, Xiangyu Zhang, Yizhuang Zhou, Jungong Han, Guiguang Ding, and Jian Sun. 2022. "Scaling up Your Kernels to 31x31: Revisiting Large Kernel Design in CNNs." *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*. https://arxiv.org/abs/2203.06717.

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.

Dwork, Cynthia, Vitaly Feldman, Moritz Hardt, Toniann Pitassi, Omer Reingold, and Aaron Roth. 2015. "Preserving Statistical Validity in Adaptive Data Analysis." *ACM Symposium on Theory of Computing*, 117–26.

Elman, Jeffrey L. 1990. "Finding Structure in Time." *Cognitive Science* 14 (2): 179–211. https:

//doi.org/10.1016/0364-0213(90)90002-E.

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.

Field, David J. 1987. "Relations Between the Statistics of Natural Images and the Response Properties of Cortical Cells." *JOSA A* 4 (12): 2379–94. https://doi.org/10.1364/josaa.4.002379.

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.

Fukushima, Kunihiko. 1980. "Neocognitron: A Self-Organizing Neural Network Model for a Mechanism of Pattern Recognition Unaffected by Shift in Position." *Biological Cybernetics* 36 (4): 193–202. https://doi.org/10.1007/BF00344251.

Gal, Yarin, and Zoubin Ghahramani. 2016. "Dropout as a Bayesian Approximation: Representing Model Uncertainty in Deep Learning." *International Conference on Machine Learning*, 1050–59. https://arxiv.org/abs/1506.02142.

Gauss, Carl Friedrich. 1809. "Theoria Motus Corporum Coelestium." In *Werke*. Königlich Preussische Akademie der Wissenschaften. https://doi.org/10.1007/978-3-642-92478-1.

Girshick, Ross, Jeff Donahue, Trevor Darrell, and Jitendra Malik. 2014. "Rich Feature Hierarchies for Accurate Object Detection and Semantic Segmentation." *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 580–87. https://doi.org/10.1109/CVPR.2014.81.

Glorot, Xavier, and Yoshua Bengio. 2010. "Understanding the Difficulty of Training Deep Feedforward Neural Networks." *Proceedings of the 13th International Conference on Artificial Intelligence and Statistics*, 249–56. https://proceedings.mlr.press/v9/glorot10a.html.

Glorot, Xavier, Antoine Bordes, and Yoshua Bengio. 2011. "Deep Sparse Rectifier Neural Networks." *Proceedings of the 14th International Conference on Artificial Intelligence and Statistics*, 315–23. https://proceedings.mlr.press/v15/glorot11a.html.

Golub, Gene H, and Charles F Van Loan. 1996. *Matrix Computations*. Johns Hopkins University Press. https://jhupbooks.press.jhu.edu/title/matrix-computations.

Goodfellow, Ian, Yoshua Bengio, and Aaron Courville. 2016. *Deep Learning*. MIT Press.

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.

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.

Grinsztajn, Léo, Edouard Oyallon, and Gaël Varoquaux. 2022. "Why Do Tree-Based Models Still Outperform Deep Learning on Tabular Data?" *Advances in Neural Information Processing Systems* 35: 507–20. https://arxiv.org/abs/2207.08815.

Guo, Chuan, Geoff Pleiss, Yu Sun, and Kilian Q. Weinberger. 2017. "On Calibration of Modern Neural Networks." *International Conference on Machine Learning*, 1321–30. https://arxiv.org/abs/1706.04599.

Hanson, Stephen José, and Lorien Y Pratt. 1988. "Comparing Biases for Minimal Network Construction with Back-Propagation." *Advances in Neural Information Processing Systems* 1.

Harris, Charles R., K. Jarrod Millman, Stéfan J. van der Walt, et al. 2020. "Array Programming with NumPy." *Nature* 585 (7825): 357–62. https://doi.org/10.1038/s41586-020-2649-2.

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.

He, Kaiming, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. "Deep Residual Learning for Image Recognition." *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 770–78. https://doi.org/10.1109/cvpr.2016.90.

He, Tong, Zhi Zhang, Hang Zhang, Zhongyue Zhang, Junyuan Xie, and Mu Li. 2019. "Bag of Tricks for Image Classification with Convolutional Neural Networks." *IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 558–67. https://arxiv.org/abs/1812.01187.

Hendrycks, Dan, and Kevin Gimpel. 2016. "Gaussian Error Linear Units (GELUs)." *ArXiv:1606.08415*. https://arxiv.org/abs/1606.08415.

Hinton, Geoffrey E., Nitish Srivastava, Alex Krizhevsky, Ilya Sutskever, and Ruslan R. Salakhutdinov. 2012. "Improving Neural Networks by Preventing Co-Adaptation of Feature Detectors." *arXiv Preprint arXiv:1207.0580*.

Hoeffding, Wassily. 1963. "Probability Inequalities for Sums of Bounded Random Variables." *Journal of the American Statistical Association* 58 (301): 13–30.

Hoerl, Arthur E, and Robert W Kennard. 1970. "Ridge Regression: Biased Estimation for Nonorthogonal Problems." *Technometrics* 12 (1): 55–67.

Holtzman, Ari, Jan Buys, Li Du, Maxwell Forbes, and Yejin Choi. 2020. "The Curious Case of Neural Text Degeneration." *International Conference on Learning Representations*. https://arxiv.org/abs/1904.09751.

Howard, Andrew G., Menglong Zhu, Bo Chen, et al. 2017. "MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications." *ArXiv:1704.04861*. https://arxiv.org/abs/1704.04861.

Hu, Jie, Li Shen, and Gang Sun. 2018. "Squeeze-and-Excitation Networks." *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 7132–41. https://doi.org/10.1109/cvpr.2018.00745.

Huang, Gao, Zhuang Liu, Laurens Van Der Maaten, and Kilian Q Weinberger. 2017. "Densely Connected Convolutional Networks." *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 4700–4708. https://doi.org/10.1109/cvpr.2017.243.

Huang, Gao, Yu Sun, Zhuang Liu, Daniel Sedra, and Kilian Weinberger. 2016. "Deep Networks with Stochastic Depth." *European Conference on Computer Vision*. https://arxiv.org/abs/1603.09382.

Hubel, David H, and Torsten N Wiesel. 1959. "Receptive Fields of Single Neurones in the Cat's Striate Cortex." *Journal of Physiology* 148 (3): 574–91. https://doi.org/10.1113/jphysiol.1959.sp006308.

Hubel, David H, and Torsten N Wiesel. 1962. "Receptive Fields, Binocular Interaction and Functional Architecture in the Cat's Visual Cortex." *Journal of Physiology* 160 (1): 106–54. https://doi.org/10.1113/jphysiol.1962.sp006837.

Hubel, David H, and Torsten N Wiesel. 1968. "Receptive Fields and Functional Architecture of Monkey Striate Cortex." *Journal of Physiology* 195 (1): 215–43. https://doi.org/10.1113/jphysiol.1968.sp008455.

Inan, Hakan, Khashayar Khosravi, and Richard Socher. 2017. "Tying Word Vectors and Word Classifiers: A Loss Framework for Language Modeling." *International Conference on Learning Representations*.

Ioffe, Sergey, and Christian Szegedy. 2015. "Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift." *International Conference on Machine Learning*, 448–56. https://arxiv.org/abs/1502.03167.

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.

Jaeger, Herbert. 2002. *Tutorial on Training Recurrent Neural Networks, Covering BPTT, RTRL, EKF and the "Echo State Network" Approach*. GMD-Forschungszentrum Informationstechnik Bonn.

Jelinek, Frederick, Robert L. Mercer, Lalit R. Bahl, and James K. Baker. 1977. "Perplexity—a Measure of the Difficulty of Speech Recognition Tasks." *Journal of the Acoustical Society*

of America 62 (S1): S63.

Keskar, Nitish Shirish, Bryan McCann, Lav R. Varshney, Caiming Xiong, and Richard Socher. 2019. "CTRL: A Conditional Transformer Language Model for Controllable Generation." *ArXiv:1909.05858*. https://arxiv.org/abs/1909.05858.

Kneser, Reinhard, and Hermann Ney. 1995. "Improved Backing-Off for M-Gram Language Modeling." *International Conference on Acoustics, Speech, and Signal Processing* 1: 181–84. https://doi.org/10.1109/ICASSP.1995.479394.

Koh, Pang Wei, Shiori Sagawa, Henrik Marklund, et al. 2021. "WILDS: A Benchmark of in-the-Wild Distribution Shifts." *International Conference on Machine Learning*, 5637–64. https://arxiv.org/abs/2012.07421.

Kohavi, Ron. 1995. "A Study of Cross-Validation and Bootstrap for Accuracy Estimation and Model Selection." *International Joint Conference on Artificial Intelligence* 14: 1137–45.

Kolmogorov, Andrey. 1933. *Grundbegriffe Der Wahrscheinlichkeitsrechnung*. Springer. https://doi.org/10.1007/978-3-642-49888-6.

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.

Krogh, Anders, and John A Hertz. 1992. "A Simple Weight Decay Can Improve Generalization." *Advances in Neural Information Processing Systems*, 950–57. https://doi.org/10.5555/2986916.2987033.

Kuzovkin, Ilya, Raul Vicente, Mathilde Petton, et al. 2018. "Activations of Deep Convolutional Neural Networks Are Aligned with Gamma Band Activity of Human Visual Cortex." *Communications Biology* 1 (1): 1–12. https://doi.org/10.1038/s42003-018-0110-y.

LeCun, Yann, Bernhard Boser, John S Denker, et al. 1989. "Backpropagation Applied to Handwritten Zip Code Recognition." *Neural Computation* 1 (4): 541–51. https://doi.org/10.1162/neco.1989.1.4.541.

LeCun, Yann, Léon Bottou, Yoshua Bengio, and Patrick Haffner. 1998. "Gradient-Based Learning Applied to Document Recognition." *Proceedings of the IEEE* 86 (11): 2278–324. https://doi.org/10.1109/5.726791.

LeCun, Yann, LD Jackel, Leon Bottou, et al. 1995. "Comparison of Learning Algorithms for Handwritten Digit Recognition." *International Conference on Artificial Neural Networks*, 53–60.

Legendre, Adrien Marie. 1805. *Mémoire Sur Les Opérations Trigonométriques: Dont Les Résultats Dépendent de La Figure de La Terre*. F. Didot.

Leshno, Moshe, Vladimir Ya. Lin, Allan Pinkus, and Shimon Schocken. 1993. "Multilayer Feedforward Networks with a Nonpolynomial Activation Function Can Approximate Any Function." *Neural Networks* 6 (6): 861–67.

Li, Mu, Tong Zhang, Yuqiang Chen, and Alexander J Smola. 2014. "Efficient Mini-Batch Training for Stochastic Optimization." *Proceedings of the 20th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, 661–70. https://doi.org/10.1145/2623330.2623612.

Li, Xiang, Shuo Chen, Xiaolin Hu, and Jian Yang. 2019. "Understanding the Disharmony Between Dropout and Batch Normalization by Variance Shift." *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2682–90. https://arxiv.org/abs/1801.05134.

Lin, Min, Qiang Chen, and Shuicheng Yan. 2013. "Network in Network." *ArXiv:1312.4400*. https://arxiv.org/abs/1312.4400.

Linnainmaa, Seppo. 1970. "The Representation of the Cumulative Rounding Error of an Algorithm as a Taylor Expansion of the Local Rounding Errors." Master's thesis, University of Helsinki.

Lipton, Zachary C, and Jacob Steinhardt. 2018. "Troubling Trends in Machine Learning Scholarship." *Communications of the ACM* 17: 45–77. https://doi.org/10.1145/3317287.3328534.

Lipton, Zachary, Yu-Xiang Wang, and Alexander Smola. 2018. "Detecting and Correcting for Label Shift with Black Box Predictors." *International Conference on Machine Learning*, 3122–30. https://arxiv.org/abs/1802.03916.

Liu, Hanxiao, Karen Simonyan, and Yiming Yang. 2018. "DARTS: Differentiable Architecture Search." *ArXiv:1806.09055*. https://arxiv.org/abs/1806.09055.

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, Zhuang, Hanzi Mao, Chao-Yuan Wu, Christoph Feichtenhofer, Trevor Darrell, and Saining Xie. 2022. "A ConvNet for the 2020s." *ArXiv:2201.03545*. https://arxiv.org/abs/2201.03545.

Long, Jonathan, Evan Shelhamer, and Trevor Darrell. 2015. "Fully Convolutional Networks for Semantic Segmentation." *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 3431–40. https://doi.org/10.1109/CVPR.2015.7298965.

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.

Lowe, David G. 2004. "Distinctive Image Features from Scale-Invariant Keypoints." *International Journal of Computer Vision* 60 (2): 91–110. https://doi.org/10.1023/b:visi.0000029664.99615.94.

Luo, Ping, Xinjiang Wang, Wenqi Shao, and Zhanglin Peng. 2018. "Towards Understanding Regularization in Batch Normalization." *ArXiv:1809.00846*. https://arxiv.org/abs/1809.00846.

Luo, Wenjie, Yujia Li, Raquel Urtasun, and Richard Zemel. 2016. "Understanding the Effective Receptive Field in Deep Convolutional Neural Networks." *Advances in Neural Information Processing Systems*. https://arxiv.org/abs/1701.04128.

Mangram, Myles E. 2013. "A Simplified Perspective of the Markowitz Portfolio Theory." *Global Journal of Business Research* 7 (1): 59–70. https://scholarworks.iu.edu/journals/index.php/jiuspa/article/view/4517.

McCulloch, Warren S, and Walter Pitts. 1943. "A Logical Calculus of the Ideas Immanent in Nervous Activity." *Bulletin of Mathematical Biophysics* 5 (4): 115–33. https://doi.org/10.1016/s0092-8240(05)80006-0.

McKinney, Wes. 2010. "Data Structures for Statistical Computing in Python." *Proceedings of the 9th Python in Science Conference*, 56–61. https://doi.org/10.25080/Majora-92bf1922-00a.

Micikevicius, Paulius, Sharan Narang, Jonah Alben, et al. 2018. "Mixed Precision Training." *International Conference on Learning Representations*.

Mikolov, Tomáš, Stefan Kombrink, Lukáš Burget, Jan Černocký, and Sanjeev Khudanpur. 2011. "Extensions of Recurrent Neural Network Language Model." *IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, 5528–31. https://doi.org/10.1109/ICASSP.2011.5947611.

Mishkin, Dmytro, and Jiri Matas. 2016. "All You Need Is a Good Init." *International Conference on Learning Representations*. https://arxiv.org/abs/1511.06422.

Moon, Taesup, Alex Smola, Yi Chang, and Zhaohui Zheng. 2010. "Intervalrank: Isotonic Regression with Listwise and Pairwise Constraints." *Proceedings of the 3rd ACM International Conference on Web Search and Data Mining*, 151–60. https://doi.org/10.1145/1718487.1718520.

Nair, Vinod, and Geoffrey E Hinton. 2010. "Rectified Linear Units Improve Restricted Boltzmann Machines." *ICML*, 807–14. https://dl.acm.org/doi/10.5555/3104322.3104425.

Nakkiran, Preetum, Gal Kaplun, Yamini Bansal, Tristan Yang, Boaz Barak, and Ilya Sutskever. 2021. "Deep Double Descent: Where Bigger Models and More Data Hurt." *Journal of Statistical Mechanics: Theory and Experiment* 2021 (12): 124003. https://doi.org/10.1088/1742-5468/ac3a74.

Ney, Hermann, Ute Essen, and Reinhard Kneser. 1994. "On Structuring Probabilistic Dependences in Stochastic Language Modelling." *Computer Speech & Language* 8 (1): 1–38. https://doi.org/10.1006/csla.1994.1001.

Nguyen, Minh Nhat, Andrew Baker, Clement Neo, Allen Roush, Andreas Kirsch, and Ravid Shwartz-Ziv. 2025. "Turning up the Heat: Min-p Sampling for Creative and Coherent

LLM Outputs." *International Conference on Learning Representations*. https://arxiv.org/abs/2407.01082.

Olshausen, Bruno A, and David J Field. 1996. "Emergence of Simple-Cell Receptive Field Properties by Learning a Sparse Code for Natural Images." *Nature* 381 (6583): 607–9. https://doi.org/10.1038/381607a0.

Ong, Cheng Soon, Alexander Smola, and Robert Williamson. 2005. "Learning the Kernel with Hyperkernels." *Journal of Machine Learning Research* 6: 1043–71. https://doi.org/10.1109/jcss.2005.25.2.

Pascanu, Razvan, Tomas Mikolov, and Yoshua Bengio. 2013. "On the Difficulty of Training Recurrent Neural Networks." *International Conference on Machine Learning*, 1310–18.

Paszke, Adam, Sam Gross, Francisco Massa, et al. 2019. "PyTorch: An Imperative Style, High-Performance Deep Learning Library." *Advances in Neural Information Processing Systems* 32: 8026–37. https://proceedings.neurips.cc/paper/2019/hash/bdbca288fee7f92f2bfa9f7012727740-Abstract.html.

Peters, Jonas, Dominik Janzing, and Bernhard Schölkopf. 2017. *Elements of Causal Inference: Foundations and Learning Algorithms*. MIT Press. https://doi.org/10.7551/mitpress/11283.001.0001.

Pleiss, Geoff, Danlu Chen, Gao Huang, Tongcheng Li, Laurens Van Der Maaten, and Kilian Q Weinberger. 2017. "Memory-Efficient Implementation of Densenets." *ArXiv:1707.06990*. https://arxiv.org/abs/1707.06990.

Popper, Karl. 2005. *The Logic of Scientific Discovery*. Routledge. https://doi.org/10.4324/9780203994627.

Power, Alethea, Yuri Burda, Harri Edwards, Igor Babuschkin, and Vedant Misra. 2022. "Grokking: Generalization Beyond Overfitting on Small Algorithmic Datasets." *ArXiv:2201.02177*. https://arxiv.org/abs/2201.02177.

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.

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.

Radosavovic, Ilija, Justin Johnson, Saining Xie, Wan-Yen Lo, and Piotr Dollár. 2019. "On Network Design Spaces for Visual Recognition." *Proceedings of the IEEE/CVF International Conference on Computer Vision*, 1882–90. https://doi.org/10.1109/iccv.2019.00052.

Radosavovic, Ilija, Raj Prateek Kosaraju, Ross Girshick, Kaiming He, and Piotr Dollár. 2020. "Designing Network Design Spaces." *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 10428–36. https://doi.org/10.1109/cvpr42600.2020.01044.

Rahimi, Ali, and Benjamin Recht. 2007. "Random Features for Large-Scale Kernel Machines." *Advances in Neural Information Processing Systems* 20.

Ranzato, Marc'Aurelio, Sumit Chopra, Michael Auli, and Wojciech Zaremba. 2016. "Sequence Level Training with Recurrent Neural Networks." *International Conference on Learning Representations*. https://arxiv.org/abs/1511.06732.

Recht, Benjamin, Rebecca Roelofs, Ludwig Schmidt, and Vaishaal Shankar. 2019. "Do ImageNet Classifiers Generalize to ImageNet?" *International Conference on Machine Learning*, 5389–400. https://arxiv.org/abs/1902.10811.

Riesenhuber, Maximilian, and Tomaso Poggio. 1999. "Hierarchical Models of Object Recognition in Cortex." *Nature Neuroscience* 2 (11): 1019–25. https://doi.org/10.1038/14819.

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.

Rumelhart, David E, Geoffrey E Hinton, and Ronald J Williams. 1986. "Learning Representations by Back-Propagating Errors." *Nature* 323 (6088): 533–36. https://doi.org/10.1038/323533a0.

Russakovsky, Olga, Jia Deng, Hao Su, et al. 2015. "ImageNet Large Scale Visual Recognition Challenge." *International Journal of Computer Vision* 115 (3): 211–52. https://doi.org/10.1007/s11263-015-0816-y.

Saerens, Marco, Patrice Latinne, and Christine Decaestecker. 2002. "Adjusting the Outputs of a Classifier to New a Priori Probabilities: A Simple Procedure." *Neural Computation* 14 (1): 21–41. https://doi.org/10.1162/089976602753284446.

Sagawa, Shiori, Pang Wei Koh, Tatsunori B. Hashimoto, and Percy Liang. 2020. "Distributionally Robust Neural Networks for Group Shifts: On the Importance of Regularization for Worst-Case Generalization." *International Conference on Learning Representations*.

Sandler, Mark, Andrew Howard, Menglong Zhu, Andrey Zhmoginov, and Liang-Chieh Chen. 2018. "MobileNetV2: Inverted Residuals and Linear Bottlenecks." *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. https://arxiv.org/abs/1801.04381.

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.

Sennrich, Rico, Barry Haddow, and Alexandra Birch. 2016. "Neural Machine Translation of Rare Words with Subword Units." *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics*, 1715–25. https://doi.org/10.18653/v1/P16-1162.

Shannon, Claude Elwood. 1951. "Prediction and Entropy of Printed English." *The Bell System Technical Journal* 30 (1): 50–64. https://doi.org/10.1002/j.1538-7305.1951.tb01366.x.

Shimodaira, Hidetoshi. 2000. "Improving Predictive Inference Under Covariate Shift by Weighting the Log-Likelihood Function." *Journal of Statistical Planning and Inference* 90 (2): 227–44.

Shwartz-Ziv, Ravid, and Amitai Armon. 2022. "Tabular Data: Deep Learning Is Not All You Need." *Information Fusion* 81: 84–90. https://doi.org/10.1016/j.inffus.2021.11.011.

Simonyan, Karen, and Andrew Zisserman. 2015. "Very Deep Convolutional Networks for Large-Scale Image Recognition." *International Conference on Learning Representations*. https://arxiv.org/abs/1409.1556.

Sivic, Josef, and Andrew Zisserman. 2003. "Video Google: A Text Retrieval Approach to Object Matching in Videos." *Proceedings of the IEEE International Conference on Computer Vision* 3: 1470–77. https://doi.org/10.1109/iccv.2003.1238663.

Smith, Samuel L., Andrew Brock, Leonard Berrada, and Soham De. 2023. "ConvNets Match Vision Transformers at Scale." *ArXiv:2310.16764*. https://arxiv.org/abs/2310.16764.

Soudry, Daniel, Elad Hoffer, Mor Shpigel Nacson, Suriya Gunasekar, and Nathan Srebro. 2018. "The Implicit Bias of Gradient Descent on Separable Data." *Journal of Machine Learning Research* 19 (70): 1–57.

Srivastava, Nitish, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. 2014. "Dropout: A Simple Way to Prevent Neural Networks from Overfitting." *Journal of Machine Learning Research* 15 (1): 1929–58. https://doi.org/10.5555/2627435.2670313.

Sutskever, Ilya, Oriol Vinyals, and Quoc V Le. 2014. "Sequence to Sequence Learning with Neural Networks." *Advances in Neural Information Processing Systems*, 3104–12. https://doi.org/10.5555/2969033.2969173.

Szegedy, Christian, Wei Liu, Yangqing Jia, et al. 2015. "Going Deeper with Convolutions." *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 1–9. https://doi.org/10.1109/cvpr.2015.7298594.

Szegedy, Christian, Vincent Vanhoucke, Sergey Ioffe, Jon Shlens, and Zbigniew Wojna. 2016. "Rethinking the Inception Architecture for Computer Vision." *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2818–26. https://doi.org/10.1109/cvpr.2016.308.

Tan, Mingxing, and Quoc Le. 2019. "EfficientNet: Rethinking Model Scaling for Convolutional Neural Networks." *International Conference on Machine Learning*, 6105–14. https://proceedings.mlr.press/v97/tan19a.html.

Tan, Mingxing, and Quoc V. Le. 2021. "EfficientNetV2: Smaller Models and Faster Training." *International Conference on Machine Learning*. https://arxiv.org/abs/2104.00298.

Telgarsky, Matus. 2016. "Benefits of Depth in Neural Networks." *Conference on Learning Theory*, 1517–39.

Tibshirani, Robert. 1996. "Regression Shrinkage and Selection via the Lasso." *Journal of the Royal Statistical Society: Series B* 58 (1): 267–88.

Touvron, Hugo, Matthieu Cord, Alexandre Sablayrolles, Gabriel Synnaeve, and Hervé Jégou. 2021. "Going Deeper with Image Transformers." *IEEE/CVF International Conference on Computer Vision*, 32–42. https://arxiv.org/abs/2103.17239.

Tsoumakas, Grigorios, and Ioannis Katakis. 2007. "Multi-Label Classification: An Overview." *International Journal of Data Warehousing and Mining* 3 (3): 1–13. https://doi.org/10.4018/jdwm.2007070101.

Vapnik, V., and A. Chervonenkis. 1971. "On the Uniform Convergence of Relative Frequencies of Events to Their Probabilities." *Theory Probab. Appl.* 16 (2): 264–81. https://doi.org/10.1137/1116025.

Vapnik, Vladimir, Esther Levin, and Yann Le Cun. 1994. "Measuring the VC-Dimension of a Learning Machine." *Neural Computation* 6 (5): 851–76. https://doi.org/10.1162/neco.1994.6.5.851.

Veit, Andreas, Michael J Wilber, and Serge Belongie. 2016. "Residual Networks Behave Like Ensembles of Relatively Shallow Networks." *Advances in Neural Information Processing Systems*. https://arxiv.org/abs/1605.06431.

Waibel, Alex, Toshiyuki Hanazawa, Geoffrey Hinton, Kiyohiro Shikano, and Kevin J Lang. 1989. "Phoneme Recognition Using Time-Delay Neural Networks." *IEEE Transactions on Acoustics, Speech, and Signal Processing* 37 (3): 328–39. https://doi.org/10.1109/29.21701.

Wengert, Robert Edwin. 1964. "A Simple Automatic Derivative Evaluation Program." *Communications of the ACM* 7 (8): 463–64. https://doi.org/10.1145/355588.365726.

Werbos, Paul J. 1990. "Backpropagation Through Time: What It Does and How to Do It." *Proceedings of the IEEE* 78 (10): 1550–60. https://doi.org/10.1109/5.58337.

Wightman, Ross, Hugo Touvron, and Hervé Jégou. 2021. "ResNet Strikes Back: An Improved Training Procedure in Timm." *ArXiv:2110.00476*. https://arxiv.org/abs/2110.00476.

Wolpert, David H. 1996. "The Lack of a Priori Distinctions Between Learning Algorithms." *Neural Computation* 8 (7): 1341–90.

Woo, Sanghyun, Shoubhik Debnath, Ronghang Hu, et al. 2023. "ConvNeXt V2: Co-Designing and Scaling ConvNets with Masked Autoencoders." *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*. https://arxiv.org/abs/2301.00808.

Wu, Yonghui, Mike Schuster, Zhifeng Chen, et al. 2016. "Google's Neural Machine Translation System: Bridging the Gap Between Human and Machine Translation." *ArXiv:1609.08144*. https://arxiv.org/abs/1609.08144.

Wu, Yuxin, and Kaiming He. 2018. "Group Normalization." *European Conference on Computer Vision*. https://arxiv.org/abs/1803.08494.

Xiao, Han, Kashif Rasul, and Roland Vollgraf. 2017. "Fashion-MNIST: A Novel Image Dataset for Benchmarking Machine Learning Algorithms." *ArXiv:1708.07747*. https://arxiv.org/abs/1708.07747.

Xie, Saining, Ross Girshick, Piotr Dollár, Zhuowen Tu, and Kaiming He. 2017. "Aggregated Residual Transformations for Deep Neural Networks." *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 1492–500. https://doi.org/10.1109/cvpr.2017.634.

Yamaguchi, Kouichi, Kenji Sakamoto, Toshio Akabane, and Yoshiji Fujimoto. 1990. "A Neural Network for Speaker-Independent Isolated Word Recognition." *First International Conference on Spoken Language Processing*. https://doi.org/10.21437/icslp.1990-282.

You, Yang, Igor Gitman, and Boris Ginsburg. 2017. "Large Batch Training of Convolutional Networks." *ArXiv:1708.03888*. https://arxiv.org/abs/1708.03888.

Yu, Fisher, and Vladlen Koltun. 2016. "Multi-Scale Context Aggregation by Dilated Convolutions." *International Conference on Learning Representations*. https://arxiv.org/abs/1511.07122.

Yun, Sangdoo, Dongyoon Han, Seong Joon Oh, Sanghyuk Chun, Junsuk Choe, and Youngjoon Yoo. 2019. "CutMix: Regularization Strategy to Train Strong Classifiers with Localizable Features." *Proceedings of the IEEE/CVF International Conference on Computer Vision*. https://arxiv.org/abs/1905.04899.

Zhang, Biao, and Rico Sennrich. 2019. "Root Mean Square Layer Normalization." *Advances in Neural Information Processing Systems* 32.

Zhang, Chiyuan, Samy Bengio, Moritz Hardt, Benjamin Recht, and Oriol Vinyals. 2021. "Understanding Deep Learning (Still) Requires Rethinking Generalization." *Communications of the ACM* 64 (3): 107–15. https://doi.org/10.1145/3446776.

Zhang, Hongyi, Moustapha Cisse, Yann N. Dauphin, and David Lopez-Paz. 2018. "mixup: Beyond Empirical Risk Minimization." *International Conference on Learning Representations*. https://arxiv.org/abs/1710.09412.

Zhang, Richard. 2019. "Making Convolutional Networks Shift-Invariant Again." *International Conference on Machine Learning*. https://arxiv.org/abs/1904.11486.

Zhang, Wei, Jun Tanida, Kazuyoshi Itoh, and Yoshiki Ichioka. 1988. "Shift-Invariant Pattern Recognition Neural Network and Its Optical Architecture." *Proceedings of Annual Conference of the Japan Society of Applied Physics*.

Zoph, Barret, and Quoc V Le. 2016. "Neural Architecture Search with Reinforcement Learning." *ArXiv:1611.01578*. https://arxiv.org/abs/1611.01578.